CA IDMS™

Database Administration Guide

Release 18.5.00, 2nd Edition

This Documentation, which includes embedded help systems and electronically distributed materials, (hereinafter referred to as the “Documentation”) is for your informational purposes only and is subject to change or withdrawal by CA at any time. This Documentation is proprietary information of CA and may not be copied, transferred, reproduced, disclosed, modified or duplicated, in whole or in part, without the prior written consent of CA.

If you are a licensed user of the software product(s) addressed in the Documentation, you may print or otherwise make available a reasonable number of copies of the Documentation for internal use by you and your employees in connection with that software, provided that all CA copyright notices and legends are affixed to each reproduced copy.

The right to print or otherwise make available copies of the Documentation is limited to the period during which the applicable license for such software remains in full force and effect. Should the license terminate for any reason, it is your responsibility to certify in writing to CA that all copies and partial copies of the Documentation have been returned to CA or destroyed.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CA PROVIDES THIS DOCUMENTATION “AS IS” WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT WILL CA BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USE OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, LOST INVESTMENT, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF CA IS EXPRESSLY ADVISED IN ADVANCE OF THE POSSIBILITY OF SUCH LOSS OR DAMAGE.

The use of any software product referenced in the Documentation is governed by the applicable license agreement and such license agreement is not modified in any way by the terms of this notice.

The manufacturer of this Documentation is CA.

Provided with “Restricted Rights.” Use, duplication or disclosure by the United States Government is subject to the restrictions set forth in FAR Sections 12.212, 52.227-14, and 52.227-19(c)(1) - (2) and DFARS Section 252.227-7014(b)(3), as applicable, or their successors.

Copyright © 2014 CA. All rights reserved. All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

CA Technologies Product References

This document references the following CA products:

- CA ACF2™ for z/OS
- CA ADS™
- CA Culprit™
- CA Endevor/DB™ for CA IDMS™ (CA Endevor/DB)
- CA IDMS™
- CA IDMS™/DC (DC)
- CA IDMS™/DC or CA IDMS™ UCF (DC/UCF)
- CA IDMS™ Performance Monitor
- CA IDMS™ Presspack
- CA IDMS™ UCF (UCF)
- CA OLQ™ Online Query for CA IDMS™ (CA OLQ)
- CA Top Secret® for z/OS

Contact CA Technologies

Contact CA Support

For your convenience, CA Technologies provides one site where you can access the information that you need for your Home Office, Small Business, and Enterprise CA Technologies products. At http://ca.com/support, you can access the following resources:

- Online and telephone contact information for technical assistance and customer services
- Information about user communities and forums
- Product and documentation downloads
- CA Support policies and guidelines
- Other helpful resources appropriate for your product

Providing Feedback About Product Documentation

If you have comments or questions about CA Technologies product documentation, you can send a message to techpubs@ca.com.

To provide feedback about CA Technologies product documentation, complete our short customer survey which is available on the CA Support website at http://ca.com/docs.

Documentation Changes

The following documentation updates were made for the 18.5.00, 2nd Edition release of this documentation:

- RESTRUCTURE Utility Statement (see page 833)—Renamed from RESTRUCTURE SEGMENT Utility Statement.
- Page Groups (see page 56)—Descriptions modified to lift restrictions on index sets spanning page group boundaries.
- DBNAME Statement (see page 165)—VERIFY ON/OFF parameter description notes describing the run time check changed to support Mixed Page Group Indexes.
- AREA Statement (see page 463)—The description of the FORCE option for the default usage mode was added.
- Default Ready Mode Using Navigational DML (see page 934)—The description of the FORCE option was added.
- Usage (see page 466)—Considerations for using the FORCE Option with ADS Dialogs were added.
- Quick Reference Information (see page 1023)—This new appendix contains reference information that was previously available in the DB Admin Quick Reference Guide.
- DISPLAY/PUNCH ALL Statement (see page 116)—The RECursive parameter that appends “AS SYNTAX.” or “AS COMMENT.” to each generated line of output was added.

Contents

Chapter 7: Physical Database DDL Statements 107

Chapter 8: Defining a Database Using SQL 223

Chapter 9: Defining a Database Using Non-SQL 237

Chapter 10: Using the Schema and Subschema Compilers 271

Chapter 11: Compiler-Directive Statements 289

Chapter 12: Operations on Entities 325

Chapter 13: Parameter Expansions 333

Chapter 14: Schema Statements 347

Chapter 16: Writing Database Procedures 523

Chapter 17: Allocating and Formatting Files 549

Chapter 18: Buffer Management 559

Chapter 19: Journaling Procedures 571

Chapter 20: Two-Phase Commit Processing 591

Chapter 21: Backup and Recovery 607

Chapter 22: Loading a Non-SQL Defined Database 673

Chapter 23: Loading an SQL-Defined Database 683

Chapter 24: Monitoring and Tuning Database Performance 701

Chapter 25: Dictionaries and Runtime Environments 723

Chapter 26: Migrating from Test to Production 747

Chapter 27: Modifying Physical Database Definitions 775

Chapter 31: Modifying Indexes, CALC Keys, and Referential Constraints 821

Chapter 32: Modifying Non-SQL Defined Databases 829

Chapter 33: Modifying Schema Entities 837

Chapter 34: Modifying Subschema Entities 865

Chapter 35: Space Management 871

Chapter 36: Record Storage and Deletion 885

Chapter 37: Chained Set Management 903

Chapter 38: Index Management 909

Chapter 39: Lock Management 929

Chapter 1: Introduction

This section contains the following topics:

Who Should Use This Guide

This guide is intended for anyone who is responsible for administering one or more CA IDMS databases and for those whose responsibility lies in administering a portion of the database, such as database definition.

Using This Guide

This guide contains all information necessary to define, load, and administer a CA IDMS database:

- **Chapter 1**—Describes who uses this guide and provides an overview of how the syntax is used.
- **Chapter 2**—Describes the CA IDMS environment.
- **Chapter 3**—Describes defining physical databases.
- **Chapter 4**—Describes defining segments, files, and areas.
- **Chapter 5**—Describes defining, generating, and punching a DMCL.
- **Chapter 6**—Discusses defining a database name table.
- **Chapter 7**—Discusses physical database DDL statements.
- **Chapter 8**—Describes defining a database using SQL.
- **Chapter 9**—Describes defining a database using non-SQL.
- **Chapter 10**—Describes using the schema and subschema compilers.
- **Chapter 11**—Discusses compiler-directive statements.
- **Chapter 12**—Discusses operations on entities.
- **Chapter 13**—Discusses parameter expansions.

- **Chapter 14**—Discusses schema statements.
- **Chapter 15**—Discusses subschema statements.
- **Chapter 16**—Discusses writing database procedures.
- **Chapter 17**—Discusses allocating and formatting files.
- **Chapter 18**—Discusses buffer management.
- **Chapter 19**—Discusses journaling procedures.
- **Chapter 20**—Discusses two-phase commit processing.
- **Chapter 21**—Discusses backup and recovery.
- **Chapter 22**—Describes loading a non-SQL defined database.
- **Chapter 23**—Describes loading an SQL-defined database.
- **Chapter 24**—Discusses monitoring and tuning database performance.
- **Chapter 25**—Describes dictionaries and runtime environments.
- **Chapter 26**—Discusses migrating from test to production.
- **Chapter 27**—Discusses modifying physical database definitions.
- **Chapter 28**—Discusses modifying database name tables.
- **Chapter 29**—Discusses modifying SQL-defined databases.
- **Chapter 30**—Describes modifying schema, view, and table definitions.
- **Chapter 31**—Discusses modifying indexes, CALC keys, and referential constraints.
- **Chapter 32**—Discusses modifying non-SQL defined databases.
- **Chapter 33**—Describes modifying schema entities.
- **Chapter 34**—Describes modifying subschema entities.
- **Chapter 35**—Describes space management.
- **Chapter 36**—Describes record storage and deletion.
- **Chapter 37**—Discusses chained set management.
- **Chapter 38**—Discusses index management.
- **Chapter 39**—Describes lock management.
- **Appendix A**—presents a sample physical database definition.
- **Appendix B**—presents a sample SQL database definition.

- **Appendix C**—presents a sample non-SQL database definition.
- **Appendix D**—Discusses native VSAM considerations.
- **Appendix E**—Discusses batch compiler execution JCL.
- **Appendix F**—Discusses system record types.
- **Appendix G**—Discusses procedures for coding a user-exit program.

Syntax Diagram Conventions

The syntax diagrams presented in this guide use the following notation conventions:

UPPERCASE OR SPECIAL CHARACTERS

Represents a required keyword, partial keyword, character, or symbol that must be entered completely as shown.

lowercase

Represents an optional keyword or partial keyword that, if used, must be entered completely as shown.

italicized lowercase

Represents a value that you supply.

lowercase bold

Represents a portion of the syntax shown in greater detail at the end of the syntax or elsewhere in the document.

◄─

Points to the default in a list of choices.

►►────────────────────

Indicates the beginning of a complete piece of syntax.

────────────────────►◄

Indicates the end of a complete piece of syntax.

────────────────────►

Indicates that the syntax continues on the next line.

►────────────────────

Indicates that the syntax continues on this line.

───────────────────►─

Indicates that the parameter continues on the next line.

─►───────────────────

Indicates that a parameter continues on this line.

►── parameter ─────────►

Indicates a required parameter.

```
►──┬─ parameter ─┬──────►
   └─ parameter ─┘
```

Indicates a choice of required parameters. You must select one.

```
►──┬─────────────┬──────►
   └─ parameter ─┘
```

Indicates an optional parameter.

```
►──┬─────────────┬──────►
   ├─ parameter ─┤
   └─ parameter ─┘
```

Indicates a choice of optional parameters. Select one or none.

```
  ┌─────────────┐
►─▼─ parameter ─┴──────►
```

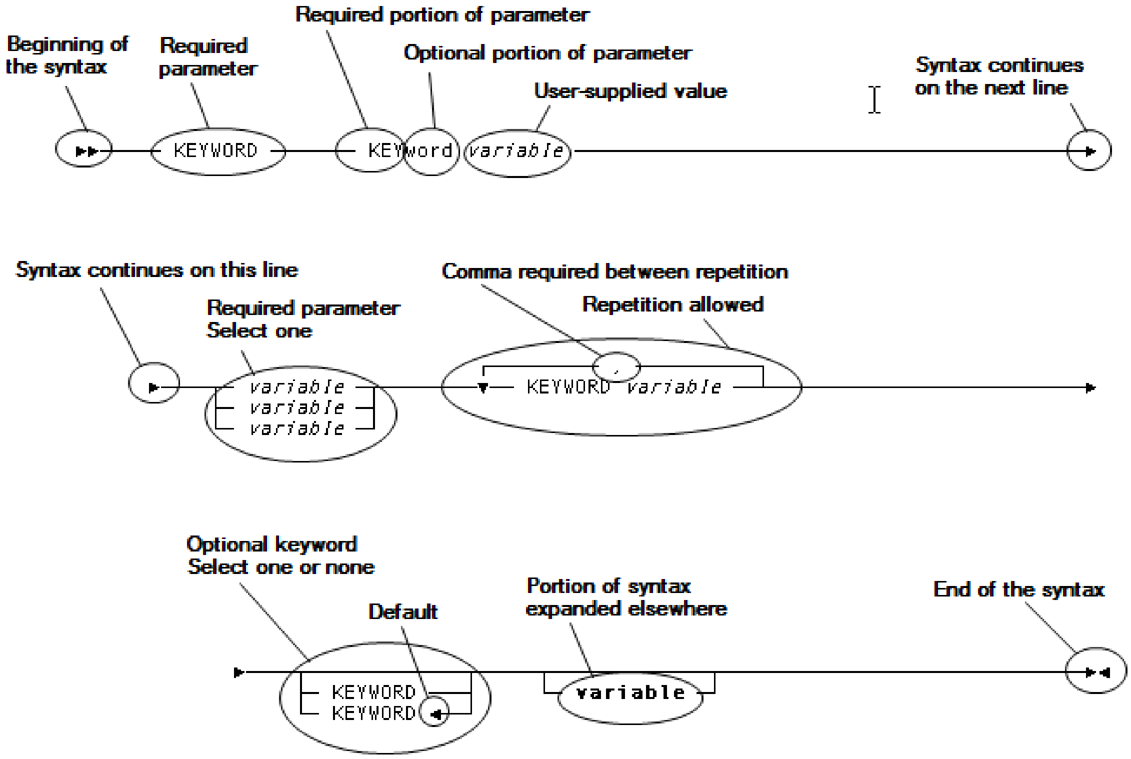
Indicates that you can repeat the parameter or specify more than one parameter.

```
  ┌──── , ─────────┐
►─▼─ parameter ────┴──────►
```

Indicates that you must enter a comma between repetitions of the parameter.

Sample Syntax Diagram

The following sample explains how the notation conventions are used:


Beginning of the syntax
Required parameter
Required portion of parameter
Optional portion of parameter
User-supplied value
Syntax continues on the next line
KEYWORD
KEYword
variable
Syntax continues on this line
Required parameter Select one
Comma required between repetition
Repetition allowed
variable
variable
variable
,
KEYWORD variable
Optional keyword Select one or none
Default
Portion of syntax expanded elsewhere
End of the syntax
KEYWORD
KEYWORD
variable

Chapter 2: CA IDMS Environment

This section contains the following topics:

Overview

CA IDMS provides both database and data communications services for the development and execution of applications in multi- and single-user environments. Development, production, and end-user systems coexist in the CA IDMS environment.

Components

CA IDMS components include the following:

- Database management system
- CA IDMS/DC or CA IDMS UCF (DC/UCF)
- Dictionaries
- Physical database definition
- Logical database definition

Types of Operation

The CA IDMS environment supports three types of operation:

- Multiuser—Implemented through CA IDMS/DC or CA IDMS UCF central version
- Single-user—Implemented through local mode
- Data Sharing—Implemented as two or more CA IDMS/DC or CA IDMS UCF central versions operating cooperatively through coupling facility services

Online programs always access the database using central version services. Batch programs can access the database either under central version or in local mode. Batch or online TP-monitor programs other than CA IDMS/DC running in another address space communicate with central version through facilities provided by CA IDMS.

Multiuser Environment

Central Version

In a multiuser environment, you use the services of the CA IDMS/DC or CA IDMS UCF central version to access the database. When two or more users attempt to access or update the database simultaneously, the DBMS, which is part of the DC/UCF system, controls and coordinates access to the database.

Central version operations provide greater concurrency and recovery services than local mode operations.

Under central version:

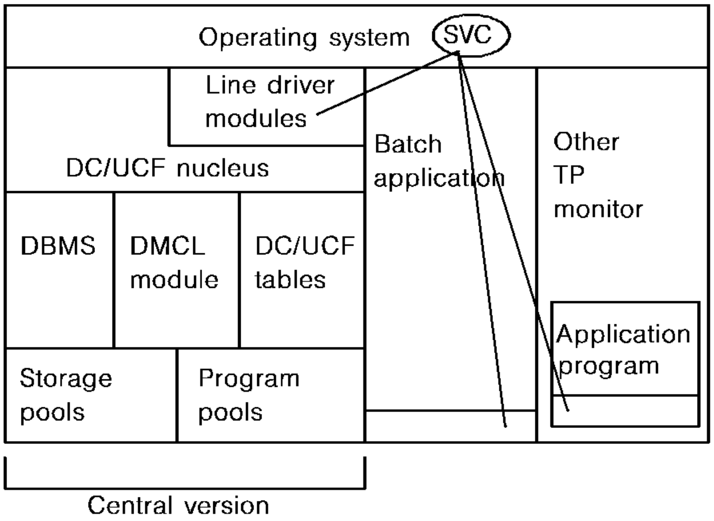
- The DBMS ensures the integrity of the database by controlling concurrent access through locks placed on areas and table rows or record occurrences.
- The DBMS performs automatic recovery operations for programs that end abnormally

Requesting Central Version Services

Application programs executing within the following environments can make database requests of the central version:

- Batch address spaces
- CA IDMS/DC and CA IDMS UCF (DC/UCF) systems
- Other teleprocessing monitors

An application program executing within the DC/UCF environment can take advantage of the single region architecture of CA IDMS. Because the database and data communications services operate within a single address space, database requests do not need to be transferred across address spaces.



Single-user Environment

Local Mode

In *local mode*, the DBMS, which is loaded at program execution time, handles requests for database services, but does not support requests from multiple users.

A batch program that runs in local mode executes entirely in its own address space.

Local mode:

- Reduces system overhead for long-running batch jobs that tend to monopolize a database area
- Controls access from concurrently executing local mode applications and central version applications through physical locks on the area. Only one address space can update an area at one time.

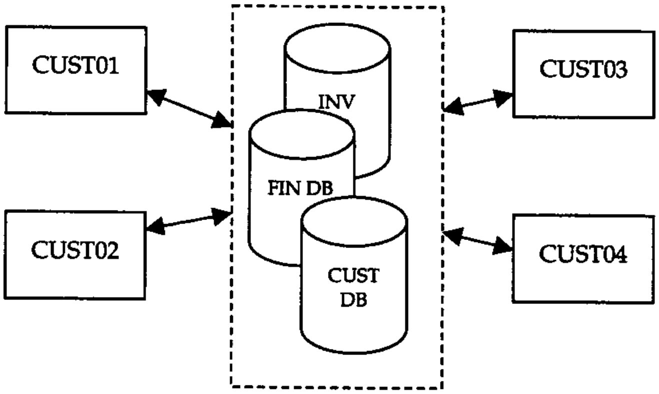
Recovery in the event of abnormal termination is accomplished through manual recovery operations.

Data Sharing Environment

A data sharing environment is one in which multiple central versions operate cooperatively through the coupling facility services of IBM's parallel sysplex architecture. Each CA IDMS/DC or CA IDMS UCF system that is to participate in a data sharing environment must be a member of a data sharing group. There can be any number of data sharing groups within a sysplex, but a central version can belong to only one group at a time.

The primary advantage of data sharing is that more than one central version can update a database concurrently. In fact, every member of a data sharing group can simultaneously update one or more databases. This enables more than one central version to service a given type of transaction, thereby providing both increased transaction throughput and fault tolerance in the event of failure.

The following diagram illustrates a data sharing group. It consists of four members (CUST01, CUST02, CUST03, and CUST04), each of which share update access to the same set of databases (Inventory, Customer, and Financial).



Note: For more information about data sharing, see the *CA IDMS System Operations Guide*.

CA IDMS/DC and CA IDMS UCF

The CA IDMS/DC system is central to the CA IDMS multiuser operating environment. CA IDMS/DC (or CA IDMS UCF) controls:

- Task management
- Terminal communications
- Scratch and queue management
- Storage and program management

Defining the System

You define the CA IDMS/DC or CA IDMS UCF system in the system dictionary through a process called *system generation* using the system generation compiler. The system definition includes:

- Definitions for system resources, programs, tasks, logs, and statistical reporting.
- Teleprocessing component definitions

Note: For more information about CA IDMS/DC and CA IDMS UCF, see the *CA IDMS System Generation Guide*.

CA IDMS/DB Components

CA IDMS/DB components include the following:

- Database management system
- Dictionaries
- Physical database definition
- Logical database definition

Database Management System

The database management system makes it possible to access the data in your database. It ensures that the data is consistent and coordinates access to data through the use of locks. The DBMS provides data integrity through automatic recovery services and has a number of tuning options such as clustering, linked lists, and data compression.

Dictionaries

What is a Dictionary

To support the runtime environment, certain information is needed to define and control that environment. This information is stored in dictionaries.

A dictionary is a special CA IDMS defined database used to hold definitions of:

- Other databases
- CA IDMS/DC or CA IDMS UCF systems
- User-written applications

There are two kinds of dictionaries used in the CA IDMS environment: system dictionaries and application dictionaries.

System Dictionary

The system dictionary contains DC/UCF system definitions and physical database definitions.

There can be only one system dictionary in a runtime environment.

Application Dictionary

An application dictionary contains application definitions and logical database definitions. This includes records, relationships, areas, schemas, subschemas, maps, and dialogs.

There can be zero, one, or more application dictionaries in a runtime environment.

Note: For more information about defining and maintaining dictionaries, see Chapter 25, "Dictionaries and Runtime Environments".

Physical Database Definition

In addition to defining the logical components of the database, you define the physical characteristics of the data and the environment in which it will be accessed. This is called the physical database definition.

The physical database definition includes the following:

- Segments, areas, and files that will hold the data
- Buffers used in retrieving and storing data
- Journal files used for recovery

The physical database definition is stored in the system dictionary, since it represents all data accessible through the runtime environment.

Logical Database Definition

The logical database definition identifies the user's view of the data.

The logical database definition includes the following:

- Definition of records, tables, and views
- Definitions of relationships between these entities
- Specification of integrity rules
- Specification of indexes and other access keys

Logical database definitions reside in the application dictionary.

Security

Access to CA IDMS databases and the DC/UCF runtime environment is controlled through a common security facility. This security facility allows access to be controlled using CA IDMS internal security services or external security packages, such as CA ACF2, CA Top Secret, or RACF.

Note: For more information, see the *CA IDMS Security Administration Guide*.

Getting Started

Installation

Before you can define, load, and access a database, you must have an operational CA IDMS environment.

To create an operational CA IDMS environment, you install CA IDMS from an integrated installation media supplied by CA. The media contains the programs and files required to install all purchased CA IDMS system software products under each supported operating system.

Note: For information about installation procedures, see the *CA IDMS Installation Guide* for your operating system.

Runtime Components

The CA IDMS runtime environment you install includes the following:

- Program libraries containing the CA IDMS/DB and CA IDMS/DC or CA IDMS UCF products
- System dictionary
- Application dictionary
- Sample database
- CA IDMS/DC or CA IDMS UCF system. This system is a starter system which you can modify to meet the needs of your environment.

Towards a Production Environment

Once you have a DC/UCF system, you are ready to define your database. The process is as follows:

1. Design the database
2. Define the database
3. Load the database
4. Develop and test applications
5. Establish the production environment

At each step you will need to:

- Establish and enforce naming conventions for entities such as schemas, database areas, records or tables, and application modules.

  A set of standardized naming conventions that suit your corporate needs will save much time and confusion and will help ensure an efficient and effective CA IDMS environment.

- Implement security measures to protect entities such as the database, data dictionary, and DC/UCF system from unauthorized access.

Designing the Database

Designing a database involves two activities:

1. Develop a design for the database
2. Decide on an implementation for that logical design

Database design is the process of determining the fundamental data entities needed to support the corporation's business.

During the initial design stage, you gather information about the business functi ons performed at your corporation. Through analysis of these functions, you identify the types of data manipulated by the functions and determine the relationships among the data types. Using data modeling techniques, you then create a diagram that serves as a logical model of the corporate data resource.

Once the initial design is complete, you enhance that design to meet specific application performance and processing requirements.

During this stage, you determine indexes and other access keys used to meet required performance goals and design structures to optimize storage resources.

Note: See the *CA IDMS Database Design Guide* for complete database design steps.

Defining the Database

At this point, you must decide on the logical definition language and translate the design into CA IDMS structures appropriate to that implementation. If you choose SQL, you must:

1. Define the physical database
2. Format the operating system files
3. Define the logical database

If you do not choose SQL, you can define the logical database either before or after defining the physical database and formatting the operating system files.

Define the Physical Database

To put the database design into effect, you set up the physical database environment. This involves identifying and sizing:

- Buffers
- Areas
- Database files
- Journal files

There is a common language used for these definitions regardless of the logical definition language chosen.

More Information

- For general information about defining the physical database, see Chapter 3, “Defining Physical Databases".
- For more information about defining the physical database, see Chapter 4, “Defining Segments, Files, and Areas” and Chapter 5, “Defining, Generating, and Punching a DMCL”.
- For more information about sizing the database, see the *CA IDMS Database Design Guide*.
- For more information about formatting operating system files, see Chapter 17, "Allocating and Formatting Files".

Define the Logical Database

Defining the logical database involves defining the data structures, such as tables and indexes, identified during the database design process. To produce this definition, you use either SQL or non-SQL statements.

More Information

- For more information about defining a logical database using SQL, see Chapter 8, “Defining a Database Using SQL".
- For more information about defining a logical database using Non-SQL, see Chapter 9, “Defining a Database Using Non-SQL".

Loading the Database

After the physical and logical database definition is complete, you load data into the database. This data may come from another database or from sequential files.

More Information

- For more information about loading the database for non-SQL defined databases, see Chapter 22, “Loading a Non-SQL Defined Database".
- For more information about loading the database for SQL defined databases, see Chapter 23, “Loading an SQL-Defined Database".

Developing and Testing Applications

After you have loaded the data into the database, you can continue to develop and test applications.

Establishing the Production Environment

When you have completed development and testing of your applications, you need to establish the production environment.

Creating Test and Production Configurations

You can set up separate configurations for test and production applications by creating:

- Two systems, two dictionaries, two databases
- One system, two dictionaries, two databases

The first approach is generally recommended in order to isolate the production environment from the impact of the test environment.

Tools for Database Definition and Maintenance

You define and maintain your database using a number of facilities.

Command Facility

The command facility is a tool used to enter:

- Physical database definition and maintenance statements
- SQL logical database definition and maintenance statements
- Utility statements

It can be run in either online or batch mode.

Note: For more information about the command facility, see the *CA IDMS Common Facilities Guide*.

Schema, Subschema, and DDDL Compilers

The batch and online schema, subschema, and data dictionary definition language (DDDL) compilers are used to define and maintain the logical definition of non-SQL databases:

- Schema compiler—Used to create a complete logical non-SQL database definition
- Subschema compiler—Used to create a subset view of the logical database definition for use with application programs.
- DDDL compiler—Used to create record and element definitions in the dictionary.

Utilities

You use utilities to perform maintenance operations on the database. Most utilities are executed as statements through the command facility; however, some are standalone programs.

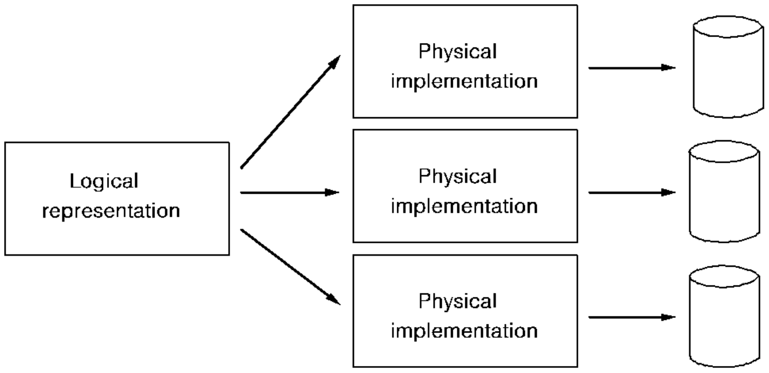
Note: For more information about utilities, see the *CA IDMS Utilities Guide*.

Chapter 3: Defining Physical Databases

This section contains the following topics:

Overview

A physical database is a collection of data that resides in operating system files. CA IDMS/DB uses information provided at runtime to determine how to map the logical representation of the database to one of perhaps many physical implementations of the database.



Physical Database Represented as Segments

The definition of a physical database is represented as a *segment*. A segment defines the areas (that is, logical files) and physical files that contain the data in the database. For CA IDMS/DB to access the segment at runtime, the segment must be added to the definition of a *DMCL*.

What is a DMCL?

A DMCL is a collection of segment definitions that can be accessed in a single execution of CA IDMS/DB. A DMCL exists as a load module in a load library and is used at runtime to determine where data required by an application is physically stored.

A DMCL also performs the following tasks:

- Assigns buffer space needed for processing the data
- Describes a buffer and files for journaling database activity
- Identifies a *database name table*, which CA IDMS/DB uses at runtime to map a logical database definition to a physical database definition
- Specifies data sharing-related attributes
- Identifies the areas of the database to be shared across members of a data sharing group

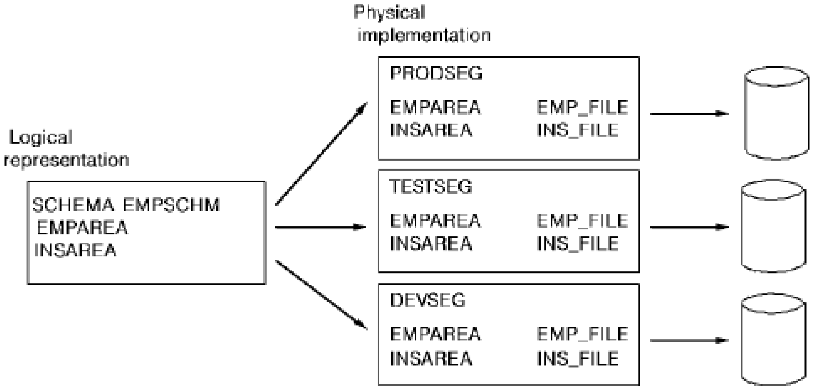
In most cases, you will need only one DMCL per configuration. For example, if you maintain separate test and production configurations, each would have its own DMCL. All applications that run under the central version use a single DMCL as specified in the system startup parameters. Applications that run in local mode can also use this DMCL.

Under local mode, you may want to use a DMCL tailored for particular applications, such as loading a database. You can specify the name of the DMCL for use in local mode in the SYSIDMS parameter file. If you do not specify a DMCL explicitly, CA IDMS/DB assumes the DMCL is named IDMSDMCL.

Segments

Represent a Physical Database

A segment represents a physical database usually defined by a single schema. It describes the collection of areas and files containing the data of the database. One logical definition (schema) can be associated with one or more physical definitions (segments). Each of these segments contains areas and files.



Areas Define Range of Database Pages

An area is a logical file divided into database pages. A database page represents a logical file block.

Database Pages Physically Stored in Files

You assign an area's pages to one or more physical disk files that exist on direct access volumes. At runtime, CA IDMS/DB maps a page in an area to one or more blocks in a file; the way CA IDMS/DB maps a database page to a physical file depends on the file's access method.

DMCLs

DMCL Contains Segments

A DMCL contains one or more segments. These may include:

- Segments that define the system dictionary
- Segments that define one or more application dictionaries
- Segments that define one or more user databases

DMCL Used at Runtime

A DMCL is the structure used by CA IDMS/DB at runtime to access physical database definitions. It must exist as a load module in a load library.

Buffers Reserve Space in Memory

A DMCL also defines two types of buffers:

- **Database buffers**, which hold database pages in use by CA IDMS/DB
- A **journal buffer** which holds journal blocks used to log database activity prior to being written to the journal file

Journal Files

Depending on your runtime environment, your DMCL will contain one of the following designations for journaling:

Environment	Journaling entities
Central version Local mode (without journaling)	■ 2 or more disk journals ■ 1 archive journal
Local mode (with journaling)	1 tape journal

Data Sharing Attributes

A DMCL used by a central version that is a member of a data sharing group also specifies attributes that are related to data sharing. These attributes include such things as the maximum number of members that can belong to the group and the action that should be taken if the coupling facility fails. These attributes are ignored by central versions that are not members of a data sharing group and by CA IDMS running in a local mode environment.

Database Name Tables

Maps Logical Definition to Physical

A database name table is an entity associated with a DMCL that is used to map the logical database definition to one or more segments in the DMCL.

Contents of a Database Name Table

The definition of a database name table includes one or more *database names*. Each database name identifies the segments to be accessed as part of the logical database. A database name table may also include one or more database group declarations.

Group Names for Dynamic Routing

In a parallel sysplex environment, a database name table may also define database groups (DBGROUPs) which represent collections of central versions to which requests can be dynamically routed. A database request can be serviced on any central version whose database name table includes the database group to which the request is directed.

Database Name Table Used at Runtime

A database name table is used by CA IDMS/DB at runtime to access physical database definitions. It must exist as a load module in a load library.

Separating Logical and Physical Database Definitions

Under CA IDMS/DB, you create a logical database definition (a schema) that contains no reference to how the data is physically stored and accessed at runtime. The physical database definition contains that information.

Advantages

The advantages of separating the logical database definition from its physical implementation are the following:

- You do not have to modify your schemas because of changes made to the physical description of a database.
- One logical database definition can have multiple physical implementations.

Before You Begin

Design the Logical and Physical Databases

Design the logical and physical database using information provided in the *CA IDMS Database Design Guide*.

Size the Physical Database

Size the database; for example, determine how large each area should be, how large the database buffers should be, and so on. You can find sizing information in the *CA IDMS Database Design Guide*.

Chapter 4: Defining Segments, Files, and Areas

This section contains the following topics:

Segments, Files, and Areas

A segment represents a physical database. It describes the physical implementation of a database whose logical contents are usually represented by a single schema. Associated with a segment are the areas and files that contain the data in the database.

Segments

The definition of a segment includes these attributes:

- What type of segment it is; that is, whether the segment definition describes the physical implementation of a non-SQL defined database or an SQL defined database
- Page groups and the maximum number of records or rows that can be stored on a database page; these two parameters determine how many pages the database can contain and the db-key format that describes the location of records or rows in the database
- For SQL-defined databases:
  - Optionally, the name of the schema for which this segment is reserved
  - Optionally, the synchronization stamp level

Example

```
create segment prodseg
  for sql
  for schema prodschm
  stamp by area;
```

Note: Segment must be added to DMCL Definition.

Before CA IDMS/DB can use a segment at runtime, you must add the segment to a DMCL, which in turn must exist as a load module in a load library.

Files

Database Files Contain Data

A CA IDMS database is stored on one or more disk files on direct access volumes. Database files contain data CA IDMS/DB accesses on behalf of applications.

What a File Defines

The definition of a file includes:

- The name of the file being defined. Within a DMCL, the name of the file must be a unique combination of the segment with which the file is associated and the file identifier.
- The type of file (that is, database or native VSAM) and the access method CA IDMS/DB is to use.
- Optionally, the data set name (or other operating system specific information) that CA IDMS/DB can use to locate the file rather than using information specified in a JCL statement.
- The external name or label in z/VSE to be used to identify the file. CA IDMS/DB searches the execution JCL for an external file name that matches the specified name and, if found, uses the JCL information to locate the dataset. If you do not specify information about the dataset in the FILE statement, you must include an external file name.

Example

```
create file emp_demo1
  assign to empfile;
```

Note: For more information about file access methods and creating files, see Chapter 17, “Allocating and Formatting Files”.

Areas

The following section Discusses related areas of database segment.

Range of Database Pages

An *area* is a contiguous range of database pages. Each page maps to one or more blocks in a file associated with the area.

Related Areas Generally Share Same Segment

Areas that contain related information are usually defined within the same segment. For example, the Commonweather database has three areas within one segment: one for employee information (EMP-DEMO-REGION), one for organizational information (ORG-DEMO-REGION), and one for benefits information (INS-DEMO-REGION).

An Area Maps to Files

Each area can map to one or more physical files. In turn, one file can contain the pages of one or more areas.

What an Area Defines

When you define an area, you assign the following attributes:

- The area's initial page range and pages reserved for future expansion
- The size of each page in the area and, optionally, a cushion reserved for expansion of variable-length records, internal index records, and compressed records and rows
- Optionally, for SQL-defined databases, whether to maintain a synchronization stamp for each table in the area or a single stamp for the entire area
- Optionally, for non-SQL defined databases, symbolic parameter values
- The file or files that contain the area's pages

Note: For more information about sizing areas and planning their use, see the *CA IDMS Database Design Guide*.

Planning

Segment Boundaries

Note the following section about segment boundaries.

One Schema One Segment

Typically one segment contains the data described by one schema. However, other factors need to be considered when deciding how data should be separated into segments.

Non-SQL Defined Data

Place all areas defined by a single schema in one segment unless:

- One or more areas are shared across multiple physical implementations. For example, if employee information is segmented by region but insurance information is corporate-wide, area(s) containing the insurance information must be placed in their own segment even though they are described in the same schema as the employee information.
- Areas defined by the schema are managed as separate units. For example, the insurance area(s) might have a different backup cycle than the employee area(s) and separating them into different segments allows operations to be performed by segment.

If areas described by a single schema are separated into different segments, it is strongly recommended that no set crosses the segment boundary (no set should have an owner in one segment and a member in another). This is advisable because it will allow you to perform maintenance operations (such as reorganization) on a segment independently of other segments. If a set crosses a segment boundary, you may need to define a new segment that includes all impacted areas or create a database name that includes all impacted segments and whose name is the same as the segment on which the operation is being performed.

SQL Defined Data

Each table is associated directly with an area in which its data rows are stored. Restrictions about where the rows of a given table can be stored are imposed by security and the DBA when defining a segment.

A segment can be reserved for tables from a specific schema by specifying the FOR SCHEMA clause on the segment definition within the application dictionary in which the tables will be defined. By specifying the FOR SCHEMA clause, the DBA ensures that only tables associated with the named schema will be stored in the segment. This can be useful in ensuring that only related production data is stored in a given segment.

In an information center or development environment in which schemas are owned by individuals, it is likely that tables from multiple schemas will reside in a single segment. Segmentation might be related more to group affiliation than to schema association.

Mapping Areas to Files

One area can be stored in multiple files and a single file can contain many areas. Typically, there is a one-to-one correspondence between an area and a file unless:

- The resulting file would be larger than a single disk device, in which case multiple files are used to contain the area
- VSAM is being used as the underlying access method and the area is larger than 4Gb, in which case the area must be mapped to multiple files
- There are a number of small, non-volatile areas, in which case multiple areas may be contained in a single file

Note: For more information about mapping areas to files, see the *CA IDMS Database Design Guide*.

Page Ranges

Areas are made up of contiguously numbered pages. The low and high page numbers assigned to an area define its page range. The page range of an area:

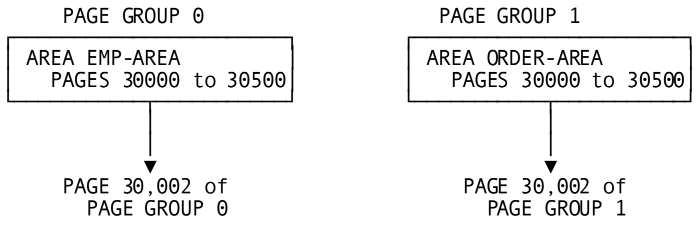
- Must not overlap that of any other area in the same segment
- Must not overlap that of any other area in a segment included in the same DMCL if the two segments have the same page group

When an area is defined, pages can be reserved for future expansion by using the MAXIMUM SPACE clause. If specified, CA IDMS ensures that no other area included in the same DMCL has a page range that overlaps both the currently allocated and the reserved space. By reserving additional pages, you are assured of being able to extend the area's page range without unloading and reloading the data.

Page Groups

Definition

A page group is an attribute of a segment. It uniquely identifies a collection of page ranges. For example, page 30,002 of page group 0 is different than page 30,002 of page group 1. The following diagram shows how page groups allow areas to be defined with the same or overlapping page ranges:



When to Use Page Groups

The default page group, 0, allows you to use up to 16,777,214 database pages containing up to 255 records/rows per page. Typically, you use page groups if your database environment requires more than 16,777,214 database pages; for example, if you access multiple, large databases within a single DMCL. By using page groups, you can include areas with the same page range in a single DMCL.

Mixed Page Groups

You may define a database with a mix of page groups; however, you may not define a database in which a chain set crosses a page group boundary. For SQL-defined databases, neither indexes nor referential constraints may cross a page group boundary.

Page Groups and Run Units

By default, a run unit can access data from only one page group at a time. This restriction can be overcome by specifying the MIXED PAGE GROUP option on the DBNAME statement that defines the database, but using this option has implications for programs accessing the database.

SQL sessions can access data in mixed page groups without any restrictions or special considerations.

Note: For more information about using mixed page groups, see the Chapter "Defining a Database Name Table".

Page Groups and Dictionaries

There are special rules about mixed page groups and dictionaries.

Note: For more information, see the Chapters "Defining a Database Name Table" and "Dictionaries and Runtime Environments".

Records Per Page

Maximum Records Per Page Affect Database Page Count

When defining a segment, you can specify the maximum number of record occurrences or rows that can be stored on a database page. The value you assign determines the db-key format, which in turn, affects the highest allowable page number that can be assigned to areas associated with the segment.

What Value Should You Use?

In most cases, use the default number of records per page, 255. This value accommodates a database with page numbers up to 16,777,214. Otherwise, choose:

- A larger value if your database contains very small records and your page size is large.
- A smaller value if your database contains very large records or you need more than 16,777,214 pages in a single database

Maximum Records Per Page Restrictions

You may define a database that has different maximum records per page for its component segments; however, you may not define a database in which components of a set, index, or referential constraint reside in areas with different maximums.

Maximum Records Per Page and Transactions

The same considerations that apply to page groups also apply to maximum records per page:

- All data accessed in a run unit must have the same maximum number of records per page, unless the MIXED PAGE GROUP BINDS option is specified on the DBNAME statement that defines the database being accessed.
- SQL transactions have no limitations in this regard.

Note: For more information about mixing maximum records per page, see Chapter 6, “Defining a Database Name Table".

Maximum Records Per Page and Dictionaries

There are special rules regarding dictionaries and maximum number of records per page.

Note: For more information, see Chapter 6, “Defining a Database Name Table” and Chapter 25, “Dictionaries and Runtime Environments”.

Page Reserve

Page reserve is the amount of space on a page that is used only for the expansion of existing records or rows. It is never used for storing new occurrences.

Specifying a page reserve as part of an area definition is useful if the area contains:

- Indexes
- Variable length records
- Compressed records or rows

The page reserve for an area can also be specified as an area override within a DMCL definition. Specifying it at the DMCL level allows tailoring the page reserve for particular types of processing, such as database loading or index building. By specifying a page reserve during these types of operations and then reducing or removing it altogether, you ensure that each page will contain free space for both new record occurrences or rows and the expansion of variable length objects.

Resolving Symbolic Parameters

Areas Resolve Schema-defined Symbols

If you defined a non-SQL schema using symbolic names for subareas, VIA-record displacement, or index attributes, you must assign values to the symbolic parameters in the physical definition of the areas.

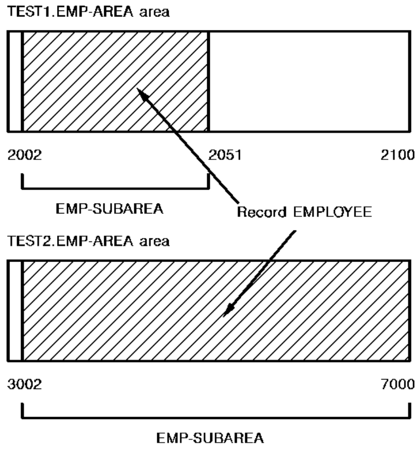
An Example of Symbolics

The following schema definition of EMPSCHM illustrates the use of a subarea symbolic. The EMPLOYEE record is stored in the EMP-SUBAREA portion of the EMP-DEMO-REGION area.

```
add schema empschm.
   add area emp-demo-region.
   add record employee                                         Logical
     location mode is calc using id-0415                       definition
     within area emp-demo-region
        subarea emp-subarea.
```

Subarea EMP-SUBAREA can be assigned different page ranges in different physical databases. For example, in segment TEST1, EMP-SUBAREA maps to pages 2002 through 2051; in segment TEST2, EMP-SUBAREA maps to pages 5002 through 7000:

```
create area test1.emp-demo-region
   primary space 100
   from page 2001
   subarea emp-subarea offset 1 for 50 percent
      .                                                         Physical
      .                                                         definition
create area test2.emp-demo-region
   primary space 2000
   from page 5001
   subarea emp-subarea offset 1 for 100 percent
      .
      .
```



Percent Specification and Area Expansion

To allow subarea page ranges to expand in proportion to increases in the area's page range, use an OFFSET specification with a percentage value in the FOR parameter. For example, the default of OFFSET 0 FOR 100 PERCENT indicates that the subarea is the entire area regardless of future expansion.

Synchronization Stamps

Table and Area Level Stamps

For SQL segments, you can specify whether synchronization stamps are to be maintained at the table or area level.

The synchronization stamp is used to make sure that the logical database definition in the access module corresponds to the current logical database definition in the dictionary.

Note: The synchronization stamp specification in the area definition included in the DMCL *must* be the same as that in the application dictionary in which the tables are defined.

At runtime, if the runtime system finds that the stamps in the access module and the database are not in sync, the access module is automatically recreated (if that option has been selected) or an error message is issued.

If you specify that the stamp is to be maintained at the table level, the stamp will be updated for an individual table when the definition of the table or any associated CALC, index, or constraint definition is modified.

If you specify that the stamp is to be maintained at the area level, the stamp will be updated when the definition of *any* table (or any associated CALC, index, or constraint definition) in the area is modified.

Which Type of Synchronization Stamp to Use

If changes to the logical structure of your database are rare (generally the case for databases in production), use area level synchronization stamps because they incur less overhead at runtime to validate. If your logical database definition changes frequently, as in a test or information center environment, choose table level synchronization stamps because a change in the definition of one table has no impact on the stamp value of other tables.

Specifying Data Set Name Information

Specifying a Data Set Name

When you access a file, you must provide information to the operating system to locate the file on disk storage. You can specify this information in one of two ways:

- In the JCL used to execute your local mode or central version system
- For z/OS and z/VM operating systems and for z/VSE with DYNAM/D, by supplying dataset information in the FILE statement

Reasons to Specify Dataset Information on the FILE Statement

The advantages of specifying the data set name or other operating system information on the FILE statement are:

- You can specify fewer statements in the system execution JCL.
- If you change the location of a file, only its definition needs changing and not every set of execution JCL.
- By not supplying an external file name (a ddname), you can ensure that only the correct file is accessed, since the dataset name cannot be overridden in the execution JCL.
- In a z/OS operating system, you can access more files if dynamic allocation is used to reference them.

Controlling the Use of Dynamic Allocation in Local Mode

By default, data set information included on the FILE statement will be used in both central version and local mode environments to dynamically allocate a data set unless the identifying information is overridden through a JCL statement.

A site may control whether dynamic allocation is used by default for local mode operations and the default behavior can be overridden for an individual job step. Both of these actions are effected through the use of the LOCAL_DYNAMIC_ALLOCATION SYSIDMS parameter. The default behavior can be established by compiling a SYSIDMS options module and it can be overridden by specifying a LOCAL_DYNAMIC_ALLOCATION parameter in the SYSIDMS file associated with the job step.

Note: For more information about the SYSIDMS parameter file, see the *CA IDMS Common Facilities Guide*.

Procedure for Defining Segments

Steps

Action	Statement
Define a segment	CREATE SEGMENT
Define one or more files to be associated with the segment	CREATE FILE
Define one or more areas to be associated with the segment	CREATE AREA
If the segment is an SQL segment, add its definition and minimally the definition of its associated areas to the application dictionary that will contain the definitions of the SQL-defined database	CREATE SEGMENT, CREATE AREA
Add the segment to an existing DMCL definition	ALTER DMCL with the ADD SEGMENT clause
Make the DMCL available to your runtime environment	See Chapter 5, “Defining, Generating, and Punching a DMCL"

Note: When copying an SQL segment definition to the application dictionary, you do not need to define the files.

Example of a Non-SQL Segment Definition

The following example creates a segment for a non-SQL defined database. The statements in the example define the segment and its associated files and areas. The characteristics of the segment are:

- Segment EMPSEG—By default, CA IDMS/DB assigns these values:
  - Page group: 0
  - Maximum records per page: 255
- File EMPDEMO1—EMPDEMO1 is a non-VSAM file with a dataset name of CORP.SYSPUB.EMPFILE1. It will be accessed using a ddname of EMPFILE1 unless overridden by a DMCL parameter

- Area EMPAREA—Area EMPAREA has the following attributes:
  - 2000 pages, starting on page 990001 and ending on page 992000. These pages will be used to store record occurrences assigned to the area. The definition does not provide for future expansion of the area because it does not specify a MAXIMUM SPACE clause.
  - Pages size of 6000 bytes.
  - A symbolic subarea, CALC-RANGE, which starts at page 990002 and extends for the remainder of the area.
  - A symbolic index, EMP-LNAME-NDX, which is a sorted index based on an index key of 10 characters and estimated entry count of 10,000
  - An association with file EMPDEMO1 that, by default, contains the entire area, beginning on block 1 of the file.

```
create segment empseg;

create file empseg.empdemo1
   assign to empfile1
   dsname 'corp.syspub.empfile1'
   disp shr;

add area empseg.emparea
   primary space 2000 pages
       from page 990001
   page size 6000 characters
   subarea calc-range offset 1 for 100 percent
   symbolic index emp-lname-ndx
       based on sorted key length 10 for 10000 records
   within file empdemo1;
```

Example of an SQL Segment Definition

The following example defines a segment and its associated areas and files for an SQL-defined database. The characteristics of the segment are:

- Segment PRODSEG—This segment has the following characteristics:
  - Is associated with SQL-schema PRODSCHM; that is, the areas in segment PRODSEG are reserved for tables in schema PRODSCHM
  - Maintains synchronization stamps at the area level (rather than the table level)
  - By default, belongs to page group 0 and contains up to a maximum of 255 rows per database page

You must define the segment in *both* the application dictionary that will contain the schema and table definitions and in the system dictionary.

- Files EMP_DEMO1 and PROJ_DEMO1—Both files are VSAM database files. At runtime, CA IDMS/DB looks in the JCL for a file specification with a matching ddname.
- Areas EMP_AREA and PROJ_AREA—The definitions of both areas allow for future expansion by using the MAXIMUM SPACE clause.

  For example, area EMP_AREA contains 1500 pages beginning on page 80001 and ending on page 81500. The first 1000 pages are the initial allocation. The remaining 500 pages are reserved for future expansion of the area. Additionally, the synchronization stamp for area EMP_AREA is by table, overriding the specification made at the segment level.

```
create segment prodseg
   for sql
   for schema prodschm
   stamp by area;

create file emp_demo1
   assign to empfile
   vsam;

create file proj_demo1
   assign to projfile
   vsam;

create area emp_area
   primary space 1000 pages
       from page 80001
   maximum space 1500 pages
   page size 6000 characters
   stamp by table
   within file emp_demo1;

create area proj_area
   primary space 1000 pages
       from page 82001
   maximum space 1500 pages
   page size 6000 characters
   within file proj_demo1;
```

More Information

- For more information about the syntax and syntax rules for the AREA, FILE, and SEGMENT statements, see Chapter 7, “Physical Database DDL Statements”.
- For more information about modifying segment definitions, see Chapter 27, “Modifying Physical Database Definitions”.
- For more information about the contents of a database page and db-keys, see Chapter 35, “Space Management".
- For more information about and a list of page number limits associated with the maximum number of records/rows per page, see 7.16, “SEGMENT Statements” in Chapter 7, “Physical Database DDL Statements”.
- For more information about sizing database areas and planning for their use, see the *CA IDMS Database Design Guide*.
- For more information about creating and formatting files, see Chapter 17, "Allocating and Formatting Files".
- For more information about loading files, see Chapter 22, “Loading a Non-SQL Defined Database” and Chapter 23, “Loading an SQL-Defined Database”.

Chapter 5: Defining, Generating, and Punching a DMCL

This section contains the following topics:

DMCLs

The DMCL is the runtime component that describes one or more physical databases. The DMCL:

- Designates which physical databases are accessible at runtime
- Describes the files used to journal database activities
- Specifies buffers for database and journal files
- Designates which areas are to be shared across members of a data sharing group
- Specifies attributes that affect data sharing operations

What a DMCL Contains

A DMCL contains the following component definitions:

Component	Function
Database buffers	Hold database pages in memory while CA IDMS/DB accesses information on the pages.
Journal buffer	Maintains information to be written to journal files, which are used for recovery operations. One and only one journal buffer must be defined for a DMCL.
Journal files	Log database activity. You can define either disk and archive journal files or a tape journal file.
Segments	Contain the areas of the database and the files to which those areas map.

DMCL Area/File Overrides

A DMCL definition can also override area and file definitions in the segments added to the DMCL.

Designating Areas as Shared

The DMCL indicates which areas are eligible to be shared for update across members of a data sharing group. Sharability can be specified for an entire segment or for an individual area through an area override.

DMCL Identifies Database Name Table

A DMCL also identifies the database name table to be used at runtime. The database name table provides logical names for one or more segments associated with the DMCL.

Order of Component Definition

To define a DMCL and its components, issue the following statements in the listed order:

1. CREATE DMCL
2. CREATE BUFFER
3. CREATE JOURNAL BUFFER
4. Either:
   - CREATE DISK JOURNAL
   - CREATE ARCHIVE JOURNAL

   Or

   - CREATE TAPE JOURNAL
5. ALTER DMCL, adding segments and optionally, any area and file overrides

DMCL Used Under the Central Version

All applications that execute under the central version use a single DMCL.

DCMLs Used in Local Mode

An application that uses local mode database services may use the same DMCL used under the central version or a DMCL tailored for local mode operations. You can define as many local DMCLs as you wish. However, generally a local mode DMCL should be created only for the following reasons:

- To execute a local mode update application with journaling activated
- To reduce core requirements in your local mode address space by reducing the number of segments in the DMCL
- To use a different page reserve or buffer size for special processing such as load operations

Differences Between Central Version and Local Mode DMCLs

The table below highlights the main differences between a DMCL used under the central version and a DMCL used only in local mode:

Component	**DMCL used under CV and in local mode**	**Local mode-only DMCL**
Buffer size	Typically large for central version operations to accommodate concurrent processing and small for local mode operations to accommodate 1 application	Typically small, to accommodate 1 application
Journal files	2 or more disk journal files and 1 or more archive files	1 tape journal file

DMCLs Used for Data Sharing

In a data sharing environment, more than one central version may share the same DMCL. If all members of a data sharing group are identical with respect to the data that they access, then they should share the same DMCL. This type of group is referred to as a homogeneous group.

If members of a group share access to only a subset of data, they may use different DMCLs. This type of group is referred to as a heterogeneous group.

The choice of whether members of a data sharing group use the same DMCL is a matter of convenience and does not affect the operation of the group. However, if different DMCLs are used, they should all specify the same data sharing attributes.

Stored as a Load Module

Because the DMCL is a runtime component, its definition must be generated and stored as a load module, and then punched and link-edited to a load library.

Identifying the DMCL to the Runtime System

You must identify the DMCL to be used in the runtime system:

- Under the central version, specify the name of the DMCL to be used as a startup parameter for the DC/UCF system. See the *CA IDMS System Operations Guide* for information about startup parameters.
- In local mode, if the name of the DMCL is not IDMSDMCL, specify the name in the SYSIDMS parameter file.

Data Sharing Attributes

What Attributes Can Be Specified?

The following data sharing-related attributes can be specified in a DMCL:

- The maximum number of members that can belong to the data sharing group
- The number of entries in the group's lock structure
- The default shared cache structure for the member using this DMCL
- The action that should be taken in response to a coupling facility failure

Group Membership

A DC/UCF system is specified to be a member of a data sharing group through parameters in the SYSIDMS file in the system's startup JCL. The system belongs to the specified group from the time it begins execution until it is shutdown. If the system abends, it remains a group member until it is restarted and terminated normally.

Specifying the Maximum Number of Members

The DMCL of each group member specifies the maximum number of members that can belong to the group at one time. The maximum number of members should be large enough to accommodate all anticipated systems, but since the value affects the size of the lock structure, it should not be larger than necessary.

What is a Lock Structure?

A lock structure is an object that resides in a coupling facility. It contains global locks that are used to control inter-member access to shared resources such as database areas and record occurrences.

Part of a lock structure is a table whose entries represent hash values. You specify the number of entries in this table as one of the data sharing attributes in the DMCL. The more entries in this table, the less likelihood there is that multiple resources will hash to the same table entry, a situation that increases locking overhead. However, the more entries in this table, the larger the lock structure needed to contain it.

Specifying the Number of Lock Table Entries

The value that you specify for the number of lock table entries should be at least as large as the highest SYSLOCKs value specified in the system definition of any member in the group. Performance may be improved by specifying an even larger value.

Note: For more information about sizing a lock structure, see the *CA IDMS System Operations Guide*.

Conflicting Group Attributes

Since the DMCL used by each member of a data sharing group specifies the maximum number of group members and the number of lock table entries, it is possible that the values specified by different members conflict. The first member to start determines the effective values and those values remain unchanged until all members of the group terminate normally. You can determine which values are in effect by issuing a DCMT DISPLAY DATA SHARING command.

What Is a Shared Cache?

A shared cache is a structure that resides in a coupling facility. It is used to contain database pages and acts as a global buffer pool shared across central versions. The use of a shared cache reduces the number of I/Os to the database.

In order to share update access to data, all files associated with a shared area must be assigned to a cache structure. One means of doing this is to specify a default shared cache for the DMCL and override the default as necessary for individual segments and files.

Note: For more information about the use of shared cache, see the *CA IDMS System Operations Guide*.

Coupling Facility Failures

In order to share update access to data, the coupling facility must be available to control access to shared resources. You may specify what action a member is to take in the event that a coupling facility structure becomes unavailable while a DC/UCF system is executing. You may direct the system to:

- Abend as soon as it detects a failure in a critical coupling facility structure
- Remain active but abend tasks that request access to shared resources

By directing the system to remain active, it can service transactions that do not access shared data. However, you will not be able to shut down the system normally since it will be unable to successfully disconnect from one or more coupling facility structures.

Note: For more information about dealing with coupling facility failures, see the *CA IDMS System Operations Guide*.

Database Buffers

What Is a Database Buffer?

A *database buffer* is space allocated in memory to hold database pages while CA IDMS/DB accesses information on those pages. A buffer is divided into *pages*. If information on the page is updated, CA IDMS/DB writes the altered page back to the database when that buffer page is needed or when the transaction ends.

CA IDMS/DB Acquires Space When It Opens Associated File

CA IDMS/DB acquires a buffer when it first opens a file associated with the buffer. If, during execution of the runtime system, CA IDMS/DB opens no files associated with the buffer, CA IDMS/DB does not acquire space for that buffer.

CA IDMS/DB Searches Buffers Before Files

To satisfy a program's request for data, CA IDMS/DB first searches the buffers to see if the requested page already resides in main memory. If the page is there, CA IDMS/DB uses the in-core copy and avoids an I/O. If it isn't, CA IDMS/DB searches the database files for the requested page.

Every File Must Be Associated with a Buffer

A database buffer must be defined to a DMCL *before* you can add segments to the DMCL definition. Each file contained in the segments added to the DMCL must be associated with a buffer. You can associate a file with a buffer in one of three ways:

- By naming the buffer in a file override added to the DMCL definition
- By naming the buffer when adding a segment to the DMCL definition
- By using the default buffer defined to the DMCL

The page size of the buffer must be greater than or equal to the block size or (in the case of VSAM) control interval of all files associated with the buffer.

What a Database Buffer Defines

The definition of a database buffer includes these attributes:

- The buffer's page size
- The number of pages in the buffer
- How CA IDMS/DB acquires storage for the buffer
- Attributes for native VSAM files

When to Define a Database Buffer

You define a database buffer when:

- You are defining a DMCL for the first time. The DMCL must have at least one database buffer.
- You have modified the database by adding another file and the anticipated use of this file indicates that another buffer will minimize contention among transactions.
- Monitoring and tuning operations indicate the need for another buffer.

Journal Buffers and Journal Files

Logs Database Activity

Journaling logs database activity on journal files. The following table describes the type of information CA IDMS/DB writes to a journal:

Type of information	Description
Database images	Contain before and after images of modified records and rows
Checkpoints	Describe a transaction event such as a COMMIT or ROLLBACK

Note: For more information about the journal records, see Chapter 19, "Journaling Procedures".

How Do You Use Journal Files?

You use the journal files to recover the database following a system or transaction failure. Typically, journaling occurs for applications that execute under the central version because CA IDMS/DB uses the journals for automatic rollback and warmstart. Journaling is less common for applications that execute in local mode, but may be used for applications that update a large database.

Note: For more information about journaling procedures under the central version and in local mode, see Chapter 19, "Journaling Procedures". Backup and recovery are discussed fully in 21.2, "Backup Procedures".

Journaling Entities

To log information about database activity, CA IDMS/DB requires the following journal entities in a DMCL:

- A *journal buffer*, which allocates space in memory to hold journal pages containing information about database activity. Each DMCL contains only one journal buffer.
- *Journal files*, to which CA IDMS/DB writes the journal pages.

When CA IDMS/DB Writes a Journal Page

CA IDMS/DB writes a journal page to the active journal file when one of the following conditions exist:

- The page in the journal buffer is full
- An update transaction terminates. A transaction terminates when the application program issues a COMMIT, COMMIT WORK, ROLLBACK, ROLLBACK WORK or FINISH statement or similar task-level statement, or when the application program aborts.
- The journal page contains before images for records or rows on a database page which must be written to the database.

Types of Journal Files

CA IDMS/DB supports the following types of journal files:

Type	Medium
Disk journal file	Disk
Archive journal file	Sequential tape or disk file (1)
Tape journal file	Sequential tape or disk file (1)

Note: (1) To be used for manual recovery, journal files on disk must be copied to tape.

Files You Choose Depend on the Runtime Environment

The type of journal files you define to a DMCL depends on whether the DMCL will be used under the central version or to journal updates made by a local mode application. A typical journaling configuration appears below:

Type of configuration	Description
Under the central version	Define: ■ 2 or more disk journals ■ 1 or more archive journals
In local mode	Define 1 tape journal

A DMCL defined with disk and archive journals can be used in local mode provided journaling is not necessary. Only a DMCL defined with a tape journal file can be used to *journal* in local mode.

Multiple Archive Files

You can define more than one archive journal. When CA IDMS/DB offloads a disk journal, it writes journal images to each archive file, thereby reducing the risk of unreadable archive journal files.

Sizing Journal Buffers

What the Journal Buffer Defines

The definition of a journal buffer defines how many pages it contains and how large the pages should be.

Buffer Page Size

The journal buffer page size determines the block size for the disk or tape journal files specified for the DMCL. Use the following criteria to choose a size for the journal buffer pages:

- If possible, the page size should be at least twice the size of the longest database record occurrence
- For VSAM disk journals, the buffer page size must be 8 bytes larger than the size of the control interval
- The page size should approximate an optimal page size for the device type in which non-VSAM disk journal files reside
- For tape journal files, the buffer page size should be as large as possible

Note: For more information about valid ranges for each operating system, see the JOURNAL BUFFER statement in Chapter 7, “Physical Database DDL Statements”.

Number of Buffer Pages

The higher the number of buffer pages, the more likely that a journal block will be found in memory eliminating the need for a disk access. Since a central version reads journal blocks primarily during rollback operations, increasing the number of journal buffer pages reduces the number of I/Os and the amount of time needed to roll out database changes.

You should minimally allocate five journal buffer pages. If you have the storage, increase this number significantly in a volatile system in which rollbacks occur frequently.

Sizing Journal Files

Disk Journal Attributes

When you define disk journal files consider the following topics:

- How many disk journal files to define
- The number of blocks in each disk journal

Number of Disk Journals

For optimal journal processing, you should define at least three disk journal files. When one file is full, CA IDMS/DB can immediately write to another file. While CA IDMS/DB writes to the alternate file, you can offload the full disk journal file to an archive file using the ARCHIVE JOURNAL utility statement. If CA IDMS/DB fills the second file, it can swap to a third file even if the ARCHIVE JOURNAL utility is still offloading the first file.

Batch Update Jobs May Require Added Files

You may need to increase the number of disk journal files when you run a batch program that updates a large volume of data. An added disk journal can prevent a situation in which the offload utility fails to complete its task before the remaining disk journal files fill.

Place Files to Avoid Offload Contention

To reduce contention during offload operations, you should:

- Place the disk journals on disk packs that do not contain database or dictionary files
- Assign each disk journal to a different volume and channel

Disk Journal File Size

The size of a disk journal file affects:

- How often the disk journal gets offloaded to the archive journal and the amount of time required to accomplish the offload. A small disk journal size means a greater number of archive tapes to keep track of since the last database backup. A large disk journal size means CA IDMS/DB will need more time to offload the disk journal to an archive file.
- The risk of losing data due to an I/O error on a journal file. A smaller file reduces the potential data loss while a larger one increases it.
- The amount of time required to perform a warmstart following a system failure. If the disk journal files are large, it may take longer for CA IDMS/DB to read through the journal in use at the time of the system failure.

  Note: You can enhance warmstart performance by using the FRAGMENT INTERVAL options of the SYSTEM system generation statement or of the DCMT VARY JOURNAL command.

Adding Segments to the DMCL

Required Segments

Segments Required for Central Version

The DMCL used under the central version must contain physical descriptions of all segments to be accessed under the central version. The segments include those defining:

- The system dictionary
- The user catalog
- Additional system areas required for central version operations:
  - DDLDCLOG
  - DDLDCRUN
  - DDLDCSCR
  - SYSMSG.DDLDCMSG
- One or more application dictionaries
- One or more user databases

The user catalog may not be required depending on your security implementation.

Note: For more information, see the *CA IDMS Security Administration Guide*.

Each central version must have its own DDLDCLOG system area and if used, its own DDLDCSCR area. In a non-data sharing environment, each central version must also have its own DDLDCRUN system area. In a non data-sharing environment, only one central version can update an area at a time. In a data-sharing environment, all areas except DDLDCLOG and DDLDCSCR can be shared and updated simultaneously by multiple central versions.

Segments Required for Local Mode

The DMCL used in local mode contains all segments to be accessed by the application. Generally these segments include:

- The system dictionary
- The user catalog
- The message area, SYSMSG.DDLDCMSG
- All user databases to be accessed by the application
- For applications using SQL:
  - A local mode scratch area, DDLOCSCR, or a system scratch area, DDLDCSCR, unless scratch in memory is in effect
  - The application dictionary containing the table definitions
- For applications using non-SQL DML:
  - The application dictionary containing the subschema load module

The system dictionary and user catalog may not be required, depending on how security is implemented in your environment.

Note: For more information, see the *CA IDMS Security Administration Guide*.

Subschema load modules can be loaded from a load library instead of a dictionary. A warning message may be written to the job log if the segment containing the load area (DDLDCLOD) for the default dictionary is not included in the DMCL.

File Limitations

Although any number of segments can be added to a DMCL, z/OS places a limit on the number of files that can be accessed within a single job step. This is a runtime restriction, since the DMCL can contain the definition of any number of files; however the number that can be accessed concurrently is limited.

Normally a z/OS job step can access up to 3,273 files. CA IDMS has extended this limit for a CV, to allow up to 10,000 files to be accessed using dynamic allocation and 3,273 files to be accessed using DD statements.

Note: Since the maximum number of DD statements that can be associated with a job step is 3273, if the number of database files in a DMCL is close to or exceeds this limit, dynamic allocation should be used for all database files so that the limit will not prevent the use of DD statements to override dynamically allocated files when necessary.

Increasing the number of files beyond the 3273 limit has implications for manual recovery, since the increased limit is supported only for CVs and not local mode batch jobs such as utility executions. To perform manual recovery, it may be necessary to execute the ROLLBACK or ROLLFORWARD utility statement multiple times, recovering a subset of the areas or segments in each execution.

Note: For more information about the impact on recovery, see 21.2, “Backup Procedures".

Area Status

Type of Access

When a DC/UCF system first accesses an area, the type of access is determined by the area status specifications within the DMCL. The choices for area status are:

- Update—indicating that database transactions executing under the central version can retrieve and update data within the area; local mode transactions and other central versions can retrieve from but not update the area.
- Retrieval—indicating that database transactions executing under the central version can retrieve but not update data in the area; a local mode transaction or another central version can update the area.
- Transient retrieval—similar to retrieval except that record (row) locks are not maintained for transactions executing within the central version.
- Offline—indicating that database transactions executing under the central version can neither retrieve nor update data in the area.

The status of an area can be changed dynamically using DCMT VARY AREA and VARY SEGMENT commands.

Retrieval Versus Transient Retrieval

Because locks are not maintained for records or rows in areas whose status is transient retrieval, less CPU (and potentially less storage) may be consumed by a transaction than if the area status were retrieval. (SQL transactions using an isolation level of transient retrieval and non-SQL transactions in a system with a sysgen specification of no retrieval locking are the exceptions.) However, an area whose status is transient retrieval must be varied offline before it can be varied to another status such as update.

To vary an area offline, all concurrently executing transactions must be terminated and all notify locks released. During the time it takes to achieve this quiesce point, new transactions will not be allowed to access the area. If this causes unacceptable processing delays the use of transient retrieval should be avoided.

Permanent Area Status

The status of an area can be changed at run time using a DCMT VARY AREA or VARY SEGMENT command. In addition to establishing a new area status, that status can also be declared as "permanent." A permanent area status remains in effect until changed by a subsequent DCMT command or until the DC/UCF system's SYSTRK or journal files are initialized. A permanent area status survives system shutdowns and abnormal terminations.

Status After System Termination

Unless a permanent area status has been established through a DCMT command, the ON STARTUP and ON WARMSTART parameters determine the status of an area when a DC/UCF system starts up. The first time a DC/UCF system is started or whenever it is restarted after a normal shutdown, the status of an area is established from the ON STARTUP specification. If the system is restarted following an abnormal termination, the status of an area is established from the ON WARMSTART specification. If the warmstart option is MAINTAIN CURRENT STATUS, the area status is set to what it was at the time of the abnormal termination.

Sharing Update Access to Data

What Is a Shared Area?

A shared area is an area that has been designated as shared. The sharability state of an area has meaning only for a central version that is a member of a data sharing group. An area that has been designated as shared can be updated concurrently by any member of the data sharing group that has access to the area in update. Only one group can have update access to an area at a time.

Designating an Area as Shared

You designate an area as shared by specifying the DATA SHARING YES clause when adding the segment to the DMCL or on a subsequent area override. The sharability state of an area can be changed at runtime by issuing a DCMT VARY SEGMENT or VARY AREA command, provided that the area's status is offline.

Shared Area Requirements

To share update access to an area, the following criteria must be met:

- All of the area's files must have an associated shared cache
- The area's characteristics must be identical in all members of the data sharing group that are to share access. These characteristics include:
  - Page range, page group, and number of records per page
  - Segment and area names
  - Page size
  - File mappings
  - IDMS file names
  - DSNAME and VOLSER of the associated disk files
- Within a data sharing group, no two shared areas can have overlapping page ranges within a page group
- Within a data sharing group, the combination of DSNAME and VOLSER must be unique for all IDMS files associated with shared areas
- A shared area cannot be native VSAM
- A shared area cannot be part of a dictionary controlled by CA Endevor/DB

If these conditions are not satisfied, you must alter your DMCL and segment definitions before declaring the area to be shared. Failure to do so will mean that one or more members of the group will be unable to access the area.

These conditions are waived on any CA IDMS system that is accessing the area in a transient retrieval mode regardless of whether the area has been designated as shared.

Area Overrides

The following information can be specified or overridden at the area level:

- Page reserve
- Central version area status
- Sharability state of an area

Overriding Page Reserve

Page reserve is space allocated on a database page for the expansion of variable-length records, bottom-level (SR8) index records, and compressed record occurrences or rows. Certain types of processing may benefit from tailored page reserves. For example, you may want to increase page reserve during an index load, after which, you reduce the page reserve.

To change the page reserve assigned to an area for a particular DMCL, override the area's definition:

```
create segment prodemp;

create area emp-area
   primary space 50 pages
   page size 1000 characters
   page reserve 0 characters
 .
 .
 .
alter dmcl idmsdmcl
   add segment prodemp
   add area prodemp.emp-area
        page reserve 250 characters;
```

After loading the index, drop the area's page reserve by dropping the area override from the DMCL definition:

```
alter dmcl idmsdmcl
   drop area prodemp.emp-area;
```

File Overrides

The following information can be specified or overridden at the file level:

- External file name (DDNAME)
- Dataset disposition for dynamic allocation
- Dataspace usage
- Buffer association
- Shared cache association

Overriding the External File Name

If your DMCL contains files defined with duplicate external file names, use the file override clause to resolve the conflict.

Dataspace Usage

Use file overrides to indicate that a file is to reside in a dataspace. If a dataspace is used, whenever a page is read from disk it will be cached in the dataspace. All future reads will receive a copy of the page in the dataspace, thus reducing I/O requests. The page will remain in the dataspace until the file is closed.

The DCMT VARY FILE command allows the dataspace specification to be changed dynamically while the system is running.

Shared Cache Association

You can associate a shared cache with a file either through a file override or by specifying a default shared cache for the file's segment. The latter is then used for all files within the segment, unless a file override specifies a different shared cache.

A default shared cache can also be specified for the DMCL. This is used only in a data sharing environment for file's whose associated area is designated as shared and for which no cache has otherwise been assigned.

Note: For more information about using shared cache, see the *CA IDMS System Operations Guide*.

Procedure for Defining a DMCL

Steps for Defining the Central Version DMCL

To create a DMCL for use under the central version, follow these steps:

Action	Statement
Create the DMCL	CREATE DMCL
Create one or more database buffers	CREATE BUFFER
Create 1 journal buffer	CREATE JOURNAL BUFFER
Create 2 or more disk journal files	CREATE DISK JOURNAL
Create 1 or more archive journal files	CREATE ARCHIVE JOURNAL
Add all segments to be used under the central version or in local mode	ALTER DMCL with the ADD SEGMENT clause
Optionally, override file or area definitions contained in segments associated with the DMCL	ALTER DMCL with the ADD FILE or ADD AREA clauses
Associate a database name table with the DMCL	ALTER DMCL with the DBTABLE clause

Example

The following example creates a DMCL to be used under the central version and in local mode. The DMCL defines one large buffer. For applications run locally, the buffer contains 100 4096-byte pages. Under the central version, the buffer initially contains 500 pages; you can increase the number of pages to 1500 dynamically by issuing a DCMT VARY BUFFER command.

```
create dmcl proddmcl dbtable proddbs;

create buffer big_buffer
   page size 4096
   local mode buffer pages 100
      opsys storage
   central version mode buffer
      initial pages 500
      maximum pages 1500
      opsys storage;

create journal buffer jrnlbuff
   page size 4096
   buffer pages 3;

create disk journal diskjnl1
   file size 1000
   assign to sysjnl1;

create disk journal diskjnl2
   file size 1000
   assign to sysjnl2;

create disk journal diskjnl3
   file size 1000
   assign to sysjnl3;

create archive journal archjrnl
   block size 16000
   assign to sysajnl1;

alter dmcl proddmcl
   default buffer big_buffer
   add segment system
   add segment defdict
   add segment empdict
   ...
   add segment empseg;
```

Steps for Defining a Local Mode DMCL

To create a DMCL for local mode only, follow the same steps as in defining a DMCL for central version use, except define a tape journal file instead of disk and archive journal files.

```
create dmcl idmsdmcl dbtable proddbs;

create buffer locl_buffer
   page size 16000
   local mode buffer pages 100
      opsys storage;

create journal buffer jrnlbuff
   page size 4096
   buffer pages 3;

create tape journal tapejnl1
   assign to tapejrnl;

alter dmcl idmsdmcl
   default buffer locl_buffer
   add segment defdict
   add segment catdict
   add segment empdict
   ...
   add segment empseg;
```

Making the DMCL Accessible to the Runtime Environment

Generate the DMCL Load Module

Generate the DMCL load module by issuing a GENERATE statement. Optionally, identify the operating system under which the DMCL will be used. For example, you can define a DMCL on a z/OS operating system that will be used under z/VM:

```
generate dmcl idmsdmcl for vm;
```

Punch the DMCL

Punch the DMCL load module using the PUNCH DMCL LOAD MODULE utility statement:

```
punch dmcl load module idmsdmcl;
```

Link-edit the DMCL

Link-edit the resulting object module to a load library using the linkage-editor for your operating system. The name under which you link the DMCL is the name by which the DMCL is known at runtime. Therefore, you can define different DMCLs and link them all with the same name provided they reside in different load libraries. This can be an advantage for local mode operations since the default DMCL used at runtime is IDMSDMCL, unless a SYSIDMS parameter is used to override the default.

Identify the DMCL to the Runtime System

Identify the DMCL to the runtime system:

- Under the central version, specify the DMCL name in the startup parameters for the DC/UCF system
- If the name of the DMCL to be used in local mode is not IDMSDMCL, identify the local mode DMCL in the SYSIDMS parameter file

More Information

- For more information about modifying DMCL definitions, see Chapter 27, "Modifying Physical Database Definitions".
- For more information about the DC/UCF system startup parameters, see the *CA IDMS System Operations Guide*.
- For more information about the SYSIDMS parameter file, see the *CA IDMS Common Facilities Guide*.
- For more information about the PUNCH utility statement, see the *CA IDMS Utilities Guide*.
- For more information about journaling procedures and offloading, see Chapter 19, “Journaling Procedures".
- For more information about buffer management and planning, see Chapter 18, “Buffer Management".
- For more information about creating disk journal files, see Chapter 17, "Allocating and Formatting Files".
- For more information about data sharing, see the *CA IDMS System Operations Guide*.
- For more information about using shared cache, see the *CA IDMS System Operations Guide*.

Chapter 6: Defining a Database Name Table

This section contains the following topics:

Overview

A database name table is used to:

- Group multiple segments under one name for processing as a single database or dictionary
- Group multiple segments under one name for maintenance operations
- Define a default dictionary for both online and local mode processing
- Identify the database to be accessed by a rununit when no database name is provided by the application or its runtime environment
- Identify the database groups to which database requests can be dynamically routed in a parallel sysplex environment

Contents of a Database Name Table

A database name table contains the definition of one or more database names defined with a CREATE DBNAME statement. Database names group segments together for processing as a single database or dictionary. Each database name definition consists of its name and the identification of one or more segments containing data required by applications accessing the named database. Additional options associated with a database name influence the processing of non-SQL applications. These options permit:

- Translating subschema names at runtime
- Restricting access to specified subschemas
- Binding a run unit to areas with a mixture of page groups and maximum records per page values

A database name table also includes a set of DBTABLE mapping rules used to identify the database or dictionary to be accessed if none is specified at runtime. These rules identify the database name to be accessed when a rununit binds to a given subschema. Every database name table must include at least one DBTABLE mapping rule to identify the default dictionary.

In a parallel sysplex environment, a database name table may also define one or more database groups defined with a CREATE DBGROUP statement. A database group represents a named collection of central versions that can service a given set of database requests. Any central version whose database name table includes the database group to which a request is directed is a member of that group and is eligible to service that request. The request will be dynamically routed to one of the CVs in the database group based on CPU availability.

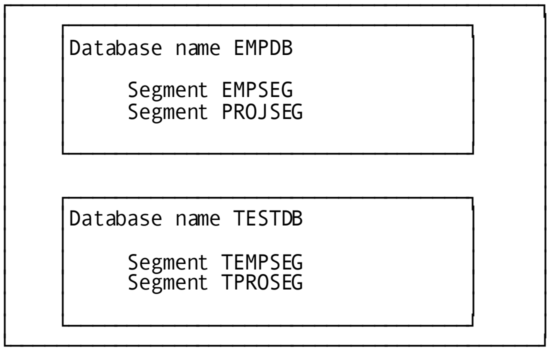
Note: For more information about DBGROUPs and dynamic routing, see the *CA IDMS System Operations Guide*.

Grouping Segments Together

The purpose of a database name is to group multiple segments together for use as a single database. Segment grouping is primarily used for defining dictionaries and non-SQL defined databases. The following example illustrates how database names can be used for defining test and production employee databases.

Each database name consists of two segments, one containing employee data and one containing project data. The production database EMPDB, contains segments EMPSEG and PROJSEG; the test database TESTDB, contains segments TEMPSEG and TPROSEG.

```
Database name table ALLDBS
```



```
Database name EMPDB

     Segment EMPSEG
     Segment PROJSEG

Database name TESTDB

     Segment TEMPSEG
     Segment TPROSEG
```

Utility Use Only Database Names

Database names can also be created simply as a means of referring to a group of segments even though no application will ever access the segments together. Creating such database names can simplify administration since certain commands, such as DCMT QUIESCE, can operate by DBNAME. In order to avoid warning messages caused by such arbitrary groups of segments, you can specify FOR UTILITY USE ONLY when defining the DBNAME.

Note: For more information about the types of warnings that may be reported, see 6.2.6, “Conflicting Names” and 6.2.7, “Mixed Page Groups and Maximum Records Per Page”.

Planning

SQL Considerations

Connecting an SQL Session

Most SQL applications will connect to the dictionary containing the definitions of the tables to be accessed. If the dictionary is composed of a single segment, no database name is required. If the dictionary is composed of multiple segments, then a database name must be created to identify all segments that make up the dictionary.

The following example shows a dictionary definition composed of three segments: a DDLDML component (testdict), a catalog component (testcat), and a message component (sysmsg):

```
 .
 .
 .
create dbname testdict
  add segment testdict
  add segment testcat
  add segment sysmsg;
```

Note: For more information about defining dictionaries, see Chapter 25, "Dictionaries and Runtime Environments".

Accessing Data through a Referencing Schema

If the SQL application accesses data through a referencing schema, the database name to which the SQL session connects may also need to include the segments containing the data to be accessed. Referencing schemas are used to provide SQL access to non-SQL defined databases and to enable different instances of an SQL-defined database to be accessed using the same table names.

When a referencing schema is defined, you can associate it with a specific database. If the referencing schema is not associated with a specific database (because its DBNAME is null), then you must include the segments containing the data to be accessed in the database name to which the SQL session connects.

In the following example, the SQL schema definition representing a non-SQL defined database does not include a DBNAME specification:

```
create schema empsql
  for nonsql schema empschm;
```

For CA IDMS to know where the non-SQL defined data is located, you must define a database name that includes both the dictionary segments and the non-SQL segments containing the data. The segment TESTCAT is the segment in which the EMPSQL schema resides. The segment TESTDICT contains the non-SQL schema EMPSCHM. The segment EMPSEG is the non-SQL segment containing the data described by schema EMPSCHM.

```
  .
  .
  .
create dbname abc
   add segment testdict
   add segment testcat
   add segment sysmsg
   add segment empseg
```

Non-SQL Considerations

Identifying Segments

When binding a rununit, the runtime system must determine which segment (or segments) contain the data to be accessed. Although the subschema identifies the areas, there may be several areas with the same name in the DMCL. To determine which area to access, the runtime system must qualify the area name with the name of a segment.

To determine the segments to be accessed, the name of a segment or database must be provided at runtime. This name can be specified in any of the following ways:

- By the application, using the DBNAME parameter on the BIND RUNUNIT statement
- From the DBNAME session attribute. Session attributes are established through user or system profiles, DCUF SET commands in DC/UCF or SYSIDMS parameters in batch
- From the DBNAME value in a SYSCTL file or an IDMSOPTI module linked with the application
- From the database name table through the use of DBTABLE mapping rules

Accessing a Single Segment

If all areas to be accessed are within one segment, the name of the segment can be specified at runtime using one of the above techniques. For example, the EMPLOAD program executed during CA IDMS installation only requires access to areas in the EMPDEMO segment. The SYSIDMS parameter file in the execution job stream specifies DBNAME=EMPDEMO, identifying the segment to be accessed. No special DBNAME entry is required.

Accessing Multiple Segments

If the application needs access to areas within multiple segments, those segments must be grouped together as a single database whose name is provided at runtime. When the bind takes place, CA IDMS locates the definition of the database in the database name table. It then searches the segments associated with that database name for a match on each area named in the subschema.

The installation process again provides an example of using a DBNAME to group segments together as one database. The system dictionary is the dictionary used to contain both physical database definitions (DMCLs, SEGMENTs, and so on) and the DC/UCF system definition. The logical name of this dictionary must be SYSTEM, since components of the runtime system access it under this name. However, it is composed of multiple segments:

- The CATSYS segment containing the DDLCAT, DDLCATX and DDLCATLOD areas
- The SYSTEM segment containing the DDLDML, DDLDCLOD and other system runtime areas
- The SYSMSG segment containing the messages issued by the runtime system

To treat all of these segments as a single database for processing by tools such as IDD and the command facility, the database name table contains a database name called SYSTEM which includes all three segments.

Using DBTABLE Mappings

When binding a rununit, if no segment or database name is explicitly established, CA IDMS searches the list of DBTABLE mapping rules in the database name table looking for one in which the "from-subschema" matches the name of the subschema specified on the bind. If a match is found, the database to be accessed is determined from the DBNAME specified in the DBTABLE mapping rule. If no match is found (and therefore no segment names can be established), the bind will fail with an error status of 1491.

To ensure that rununits will bind successfully, you must specify DBTABLE mappings for all rununits that bind without establishing a DBNAME. For example, if all rununits binding to a subschema whose name begins with INS are to access the insurance database INSDB then specify the following DBTABLE mapping:

```
alter dbtable alldbs
  subschema ins????? maps to ins????? dbname insdb;
```

Using Subschema Mappings

When defining a database name, you can specify subschema mapping rules that change the name of the subschema specified by the application at the time a rununit is bound. This feature allows an application program to be compiled against one subschema but execute using a different subschema. This can be useful when:

- The two subschemas are derived from different schemas (for example, test and production schemas)
- The two subschemas are derived from different versions of the same schema (a change was made to the schema and new subschemas created)

For example, two schemas, EMPSCHM and TEMPSCHM define the production and test versions of the same database. Separate schemas are maintained so that changes can be made to the test version without impacting production. Each schema has a set of subschemas: EMPPxxxx are production subschemas and EMPTxxxx are test subschemas. Programs are compiled against the test subschemas and copied into the production libraries when ready. The following subschema mapping rule ensures that rununits use production subschemas when binding to the production (EMPDB) database:

```
    create dbname alldbs.empdb
      subschema empt???? maps to empp????
.
.
.
```

Additional Segments

In your database name definition, you must identify the segments containing the data to be accessed by applications binding to that database. If all applications specify a DBNAME on the BIND RUNUNIT statement, then only segments accessed by those rununits need to be included in the database name. If, on the other hand, the database name is specified externally (for example by using a DCUF command), then you may need to include additional segments within your database name definition or use subschema mapping rules to ensure that rununits bind successfully.

To illustrate this, assume that an application requires access to both employee and project data in segments EMPSEG and PROJSEG respectively. To satisfy this application, a database name of EMPDB is created:

```
create dbname alldbs.empdb
  include segment empseg
  include segment projseg;
```

Instead of specifying EMPDB within the application, the user issues a DCUF SET DBNAME command to establish EMPDB as the DBNAME session attribute. Because this session attribute applies to all rununits initiated on behalf of the user, to satisfy another rununit accessing insurance information, either:

- Include the insurance segment in the EMPDB database name

  or

- Use subschema mapping rules on the DBNAME statement for EMPDB to redirect the insurance rununit to a different database name:

  ```
  alter dbname alldbs.empdb
    subschema empt???? maps to empp????
    subschema ???????? uses dbtable mapping;
  ```

In addition to changing the name of employee subschemas, these parameters have the effect of treating rununits binding to other subschemas, as if no DBNAME were specified; instead, the database name is selected using the DBTABLE mapping rules.

Restricting Subschema Names

Determines Valid Subschemas

You can request that the subschema name bound by an application be present in the database name table in order for the application to execute using the subschema. You can use this feature to prevent access to the database from an unauthorized subschema.

To request this feature, specify MATCH ON SUBSCHEMA REQUIRED on the DBNAME statement.

Note: You can also achieve the same or better protection using rununit security documented in *CA IDMS Security Administration Guide*.

Application Dictionaries

Database Name Required

In most cases, each application dictionary will require a separate database name definition. The only time a database name definition is not required for a dictionary is if all areas other than the system message area are in one segment.

Note: For more information about defining dictionaries, see Chapter 25, “Dictionaries and Runtime Environments".

Sharing Areas

If areas are shared between dictionaries, place those areas in separate segments and include the segment in all appropriate database names. For example, if two or more dictionaries share the same DDLDCLOD area, then place the load area in its own segment and define a database name for each dictionary including in each the segment that contains the shared load area.

Mixed Page Groups

If your dictionaries have different page groups (or maximum records per page), they cannot share areas. This also applies to the system message area, which can be included only in dictionaries with the same page group.

Note: All of the AREAS in a dictionary subschema (for example, IDMSNWKA, IDMSNWKG, and so on) must be in the same PAGE GROUP. A BIND for such a subschema containing AREAS that map to different PAGE GROUPS is not supported and will result in the task being abended with error messages DB347030 and DC208001 and abend code 5007.

Defining the Default Dictionary

What Is a Default Dictionary?

The default dictionary is the dictionary accessed by SQL applications, CA IDMS tools and other runtime components when none is specified through other means. For example, if the DDDL compiler is executed in batch and a dictionary is neither specified on a SIGNON statement nor a SYSIDMS parameter, the default dictionary is accessed.

Defining a Default Dictionary

The default dictionary is defined using a DBNAME statement and is identified as the default by a DBTABLE mapping rule.

By convention, the default dictionary is identified (using DBTABLE mapping rules) as the database name to which the IDMSNWKL subschema maps. Since all subschemas whose names begin with "IDMSNWK" are typically mapped in the same way, the DBTABLE mapping rule defining the default dictionary usually specifies IDMSNWK? as a subschema name.

The following statements define TESTDICT, which is comprised of segments TESTDICT and SYSMSG, as the default dictionary for the ALLDBS database name table:

```
  create dbtable alldbs
    subschema idmsnwk? maps to idmsnwk? dbname testdict
.
.
.
  create dbname alldbs.testdict
    segment testdict
    segment sysmsg;
```

Every database name table must have a default dictionary specification.

Conflicting Names

Area Names

If you have database areas with conflicting names, you must define separate database names for each set of conflicting areas. This means that if two segments have an identically named area, they cannot be included within the same database name. Areas that must be shared across databases (for example, areas containing corporate-wide insurance information) should be placed in their own segment so that they can be included in multiple database names without causing conflicts.

Segment and Database Names

If a DMCL includes a segment with the same name as a database in the associated database name table, then that database name must include the segment of the same name. For example, if a DMCL contains a segment named EMPDB and its associated database name table contains a database name called EMPDB, then the segment EMPDB must be included in the database named EMPDB. This ensures that applications accessing EMPDB will always access the same data.

Checking for Conflicts

Both of the above conditions are checked by the runtime system. If a name conflict is detected, the database name is flagged in error and no application will be able to access it. To detect conflicts before placing a new DMCL or database name table into production, use the DMCL option of the IDMSLOOK utility.

To eliminate warning messages for database names created only for administrative convenience, you can designate them for utility use only.

Mixed Page Groups and Maximum Records Per Page

What Is Allowed?

SQL access to data in mixed page groups and with different maximum records per page is always allowed. However, by default, CA IDMS does not support the ability to access data in areas with different page groups or maximum records per page from a run unit. Therefore, if you need to access a database which exceeds the size limits of a single page group or which uses different record maximums from a single run unit, you must indicate this by specifying the MIXED PAGE GROUP BINDS ALLOWED option on the DBNAME statement that defines the database.

What Happens When Binding a Run Unit?

If an application program binds a run unit to a database that includes segments with a mix of page groups or maximum records per page, the bind may or may not succeed depending on the MIXED PAGE GROUP option specified on the database's DBNAME statement:

- MIXED PAGE GROUP BINDS ALLOWED—The bind will succeed regardless of what areas are included in the subschema.
- MIXED PAGE GROUP BINDS NOT ALLOWED—The bind will succeed only if all areas in the subschema are in the same page group and have the same maximum records per page.

Detecting Potential Problems

You can detect potential problems ahead of time by using the IDMSLOOK utility (or the LOOK system task). The DMCL option will warn you if you have mixed page groups or maximum records per page within any of your database names. The BIND option will indicate whether a bind run unit will succeed for a specified subschema and database.

To eliminate warning messages for database names created only for administrative convenience, you can designate them for utility use only.

Application Program Considerations

Special care must be taken in navigational-DML application programs that access data with a mix of page groups or maximum records per page. If the application program retrieves a record by dbkey then it must do one of the following:

- Specify on the DML command the name of the record that it is trying to retrieve
- Specify on the DML command the page group and maximum records per page of the record that it is trying to retrieve
- Ensure that the current page group and maximum records per page are correct for the record that it is trying to retrieve. The current page group and record maximum are those associated with the dbkey that is current of run unit.

Failure to take one of these actions may lead to the inability to retrieve any record or the retrieval of unintended records.

Identifying Potential Problem Programs

Numbered exit, Exit 34, is provided for use with the MIXED PAGE GROUP BINDS ALLOWED option. You can use this exit to help identify applications that may require modification to function correctly when mixed page group support is enabled.

Note: For more information about Exit 34, see the *CA IDMS System Operations Guide*.

Dictionary Considerations

MIXED PAGE GROUP BINDS ALLOWED cannot be specified for dictionaries. When defining a dictionary with a mixture of page groups or maximum records per page, the following rules must be observed:

- The DDLDML and DDLDCLOD areas must be in the same page group and have the same maximum records per page. The DDLDCMSG area (if included in the DBNAME) must also have the same page group and record maximum.
- The DDLCAT, DDLCATX, and DDLCATLOD areas must be in the same page group and have the same maximum records per page.
- These two area sets may be in different page groups.
- Dictionaries that share load areas must be in the same page group.

If you define a dictionary with a mixture of page groups or maximum records per page, certain utility functions such as UNLOAD can only be performed by segment or individual area, rather than for the dictionary as a whole.

Sharing Database Name Tables

One Database Name Table Per Environment

In most cases only one database name table is needed for each of your runtime environments. This means that all DMCLs defined in a system dictionary normally specify the same database name table in their DBTABLE clause. The database name table used at runtime is the one identified in the DMCL being used.

Missing Segments

Since multiple DMCLs are associated with the same database name table, it is possible (in fact likely) that a segment included in a database name is not included in the DMCL being used. This is a normal condition and will result in an error only if an application attempts to access data from the missing segment.

Defining and Generating the Database Name Table

Steps to Follow

Define and generate the database name table using the steps listed as follows.

The database name table must exist as a module in a load library in order to be usable by the runtime system. The name of the load module assigned in the link-edit must match the name specified in the DMCL.

Action	Statement
Create the database name table, adding DBTABLE mappings to define a default dictionary and for non-SQL applications binding without a DBNAME	CREATE DBTABLE
Create the database names, adding the segments and subschema mappings required by your applications	CREATE DBNAME
Generate the database name table	GENERATE DBTABLE
Associate the database name table with a DMCL	ALTER DMCL
Punch the database name table load module and link-edit it to a load library	PUNCH DBTABLE LOAD MODULE

Example

The example below defines a basic database name table that is suitable if all non-dictionary segments in your runtime environment are in the same page group and have unique area names.

It has the following characteristics:

- A database name for each of the following dictionaries: an application dictionary called DEFDICT, the system dictionary called SYSTEM and the SYSDIRL dictionary containing report definitions and dictionary schemas.
- A DBTABLE mapping rule identifying DEFDICT as the default dictionary
- A database name called DEFDB that includes all non-SQL defined segments (other than those related to a dictionary)
- A DBTABLE mapping rule identifying DEFDB as the database to be accessed by all non-SQL applications that do not specify a DBNAME

```
      create dbtable alldbs
       subschema idmsnwk? maps to idmsnwk? dbname defdict
       subschema ???????? maps to ???????? dbname defdb;

      create dbname system
       segment catsys
       segment system
       segment sysmsg;
      create dbname defdict
       segment defdict
       segment defcat    ◄-- for SQL users
       segment sysmsg;
      create dbname defdb
       segment user-segment1
       segment user-segment2
.
.
.
      generate dbtable alldbs;
```

More Information

- For more information about modifying the database name table, see Chapter 28, “Modifying Database Name Tables".
- For more information about establishing the runtime environment and defining dictionaries, see Chapter 25, "Dictionaries and Runtime Environments".
- For more information about system generation, see the *CA IDMS System Generation Guide*.
- For more information about DCMT commands, see the *CA IDMS System Tasks and Operator Commands Guide*.
- For more information about DBGROUPs and dynamic routing, see the *CA IDMS System Operations Guide*.

Chapter 7: Physical Database DDL Statements

This section contains the following topics:

Statement Summary

Physical Database Description Statements

The following table summarizes the statements described in this chapter in order by verb. The statement descriptions are arranged in alphabetic order by noun.

Statement	Purpose
ALTER ARCHIVE JOURNAL	Modifies the definition of an archive journal file
ALTER AREA	Modifies the definition of an area
ALTER BUFFER	Modifies the definition of a database buffer
ALTER DBGROUP	Modifies a database group within a database name table
ALTER DBNAME	Modifies an entry in the database name table
ALTER DBTABLE	Modifies a database name table definition

Statement	Purpose
ALTER DISK JOURNAL	Modifies the definition of a disk journal file
ALTER DMCL	Modifies a DMCL definition
ALTER FILE	Modifies the definition of a database file
ALTER JOURNAL BUFFER	Modifies the definition of a journal buffer
ALTER SEGMENT	Modifies the definition of a segment
ALTER TAPE JOURNAL	Modifies the definition of a tape journal file
CREATE ARCHIVE JOURNAL	Defines an archive journal file
CREATE AREA	Defines an area
CREATE BUFFER	Defines a database buffer
CREATE DBGROUP	Adds a database group to a database name table
CREATE DBNAME	Adds an entry to the database name table
CREATE DBTABLE	Creates a database name table
CREATE DISK JOURNAL	Defines a disk journal file
CREATE DMCL	Defines a DMCL
CREATE FILE	Defines a database file
CREATE JOURNAL BUFFER	Defines a journal buffer
CREATE SEGMENT	Defines a segment
CREATE TAPE JOURNAL	Defines a tape journal file
DISPLAY ARCHIVE JOURNAL	Displays the definition of an archive journal file
DISPLAY AREA	Displays the definition of an area
DISPLAY BUFFER	Displays the definition of a database buffer
DISPLAY DISK JOURNAL	Displays the definition of a disk journal file
DISPLAY DMCL	Displays a DMCL definition
DISPLAY FILE	Displays the definition of a database file
DISPLAY JOURNAL BUFFER	Displays the definition of a journal buffer
DISPLAY SEGMENT	Displays the definition of a segment
DISPLAY TAPE JOURNAL	Displays the definition of a tape journal file
DROP ARCHIVE JOURNAL	Deletes the definition of an archive journal file
DROP AREA	Deletes the definition of an area
DROP BUFFER	Deletes the definition of a database buffer

Statement	Purpose
DROP DBGROUP	Deletes a database group from the database name table
DROP DBNAME	Deletes an entry from the database name table
DROP DBTABLE	Deletes the definition of a database name table
DROP DISK JOURNAL	Deletes the definition of a disk journal file
DROP DMCL	Deletes a DMCL definition
DROP FILE	Deletes the definition of a database file
DROP JOURNAL BUFFER	Deletes the definition of a journal buffer
DROP SEGMENT	Deletes the definition of a segment
DROP TAPE JOURNAL	Deletes the definition of a tape journal file
GENERATE DBTABLE	Generates a database name table load module
GENERATE DMCL	Generates a DMCL load module
PUNCH ARCHIVE JOURNAL	Punches the definition of an archive journal file
PUNCH AREA	Punches the definition of an area
PUNCH BUFFER	Punches the definition of a database buffer
PUNCH DISK JOURNAL	Punches the definition of a disk journal file
PUNCH DMCL	Punches a DMCL definition
PUNCH FILE	Punches the definition of a database file
PUNCH JOURNAL BUFFER	Punches the definition of a journal buffer
PUNCH SEGMENT	Punches the definition of a segment
PUNCH TAPE JOURNAL	Punches the definition of a tape journal file

Components of a Physical DDL Statement

Keywords, Values, and Separators

Physical DDL statements consist of:

- *Keywords*
- *User-supplied values* that:
  - Identify specific occurrences of entities (for example, the EMP_BUFF database buffer)
  - Specify data values (for example, 983 or 'Boston')
- *Separators* that separate keywords and user-supplied values from one another. A separator can be a space, a comment, or a new-line character (for example, [Enter]).

Where Separators Are Not Required

Separators are *not* required before or after a character string literal or any of the following symbols:

Symbols	Description
:	Colon
,	Comma
.	Period
;	Semicolon

Clauses in Syntax Statements Are Not Positional

The clauses in the syntax statements that appear in this chapter are *not* positional. That is, you can code the clauses in any order.

Verb Synonyms

The following table summarizes synonyms for the verbs CREATE, ALTER, DROP, and ADD:

Verb	Synonym
CREATE	ADD
ALTER	MODIFY, MOD

Verb	Synonym
DROP	DELETE, DEL when part of the main syntax statement EXCLUDE, EXC when part of a clause
ADD	INCLUDE, INC

Naming Conventions

Valid Characters

A physical DDL entity name consists of a combination of:

- Upper case letters (A through Z)
- Digits (0 through 9)
- At sign (@)
- Dollar sign ($)
- Pound sign (#)
- Hyphen (-) or underscore (_), but not both; do *not* use a hyphen or underscore when naming the following entities:
  - DBNAME
  - DBTABLE
  - DMCL
  - SEGMENT

The first character of an identifier must be a letter, @, $, or #. If you like, you can enclose the identifier in double quotes (").

Qualifying Entity Names

Names for some entities can be qualified by names of other entities. For example, a database buffer can be qualified by the name of the DMCL with which it is associated.

To qualify an entity, specify the qualifier first, followed by a period (.), followed by the name of the entity you are qualifying. For example, the following qualified identifier identifies the EMP_BUFF database buffer associated with DMCL IDMSDMCL:

```
idmsdmcl.emp_buff
```

Number of Characters

The following table summarizes how long each entity name can be:

Maximum Length	Physical Database Entity
18	ARCHIVE JOURNAL AREA BUFFER DISK JOURNAL FILE JOURNAL BUFFER TAPE JOURNAL
8	DBNAME DBTABLE DMCL SEGMENT

Using Lowercase Letters in Identifiers

Some physical DDL statements contain references to SQL entities. For example, you can specify the name of an SQL schema on a SEGMENT statement. If the schema name is case sensitive, enclose it in double quotes:

```
for sql schema "Devschm"
```

If you code other physical DDL entities in lower case letters, CA IDMS/DB automatically converts them to upper case.

Keywords as Identifiers

Why Avoid Keywords as Identifiers

The use of keywords as identifiers can cause ambiguity in some circumstances. You should therefore avoid using keywords as identifiers.

If you must use a keyword as an identifier, enclose the identifier in double quotation marks to prevent possible ambiguity.

Note: For information about submitting physical DDL statements to the command facility, see the *CA IDMS Common Facilities Guide*.

Entity Currency

Entities That Establish Currency

The DMCL, SEGMENT, and DBTABLE entities establish *currency* for associated entities, as shown in the following table:

Current Entity	Associated Entities
DMCL	ARCHIVE JOURNAL BUFFER DISK JOURNAL JOURNAL BUFFER TAPE JOURNAL
SEGMENT	AREA FILE
DBTABLE	DBNAME

How Is Currency Established?

Currency is established when you:

- Perform a CREATE or ALTER operation on a DMCL, SEGMENT, or DBTABLE entity occurrence
- Fully qualify the name of an entity associated with a DMCL, segment, or database name table on a CREATE, ALTER, or DROP statement

Subsequent operations on associated entities are applied to that particular DMCL, segment, or database name table. The following example establishes IDMSDMCL as the current DMCL occurrence. The database buffer statement that follows implicitly associates the named buffer with IDMSDMCL:

```
alter dmcl idmsdmcl;

create buffer index_buffer
  page size 1076
  local mode buffer pages 10
  central version buffer
    initial pages 100
    maximum pages 500;
```

Use Fully-qualified Names if Currency Not Established

If you don't establish currency on a DMCL, segment, or database name table before operating on an associated entity, you must qualify the name of the associated entity with the name of the DMCL, segment, or database name table. In the following example, the BUFFER statement must qualify the named buffer with the name of the DMCL because DMCL currency was not first established:

```
create buffer idmsdmcl.index_buffer
  page size 1076
  local mode buffer pages 10
 .
 .
 .
```

Once this statement is executed, IDMSDMCL is established as the current DMCL.

Generic DISPLAY/PUNCH Statement

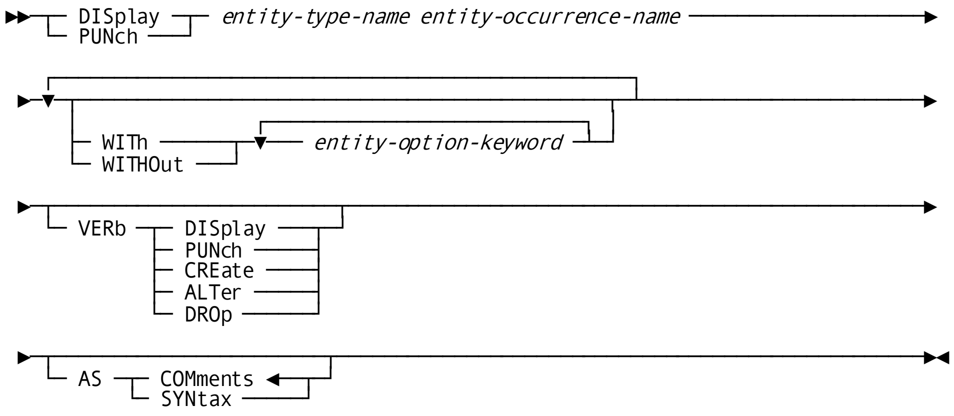
The DISPLAY and PUNCH operations produce as output the DDL statements that describe the named entity. DISPLAY and PUNCH operations do not update the entity description.

The location of the output depends on which verb is used and whether you are using the online or the batch command facility:

- *DISPLAY* displays online output at the terminal and lists batch output in the command facility's activity listing.
- *PUNCH* writes the output to the system punch file. All punched output is also listed in the command facility's activity listing.

Syntax

Display and punch statements share common clauses. Syntax descriptions for these common clauses appear below. Deviations from these descriptions for particular entities appear in the syntax description for that entity.



Parameters

entity-type-name

Identifies the type of entity to display or punch.

entity-occurrence-name

Specifies the name of the entity occurrence to display or punch. If there is no current associated entity, *entity-occurrence-name* must be the fully qualified name of an existing occurrence of the specified entity type. For example, to display an area, you must either qualify the area with the name of its associated segment or obtain currency on that segment before issuing the DISPLAY AREA statements.

WITh

Displays or punches only the parts of the entity description specified by *entity-option-keyword* in addition to parts that are always included, such as the entity occurrence name.

WITHOut

Does *not* display or punch the specified options. *Other* options in effect through the WITH clause in the current DISPLAY statement are displayed.

entity-option-keyword

Specifies options to display or punch. *Entity-option-keyword* differs for each entity. See the description of a particular entity for more information.

VERB

Specifies the verb with which the entity statement is to be displayed or punched. For example, if VERB CREATE is specified, the output of the DISPLAY/PUNCH statement is a CREATE statement. If VERB DROP is specified, the output is a DROP statement, and so on. If this clause is not coded, the verb used is the one shown as the default in the syntax diagram for the specific entity being displayed.

AS COMments

Outputs physical database syntax as comments with the characters *+ preceding the text of the statement. AS COMMENTS is the default.

AS SYNtax

Outputs physical database syntax which can be edited and resubmitted to the command facility.

Usage

Code Only One WITH Clause

Only one WITH clause is permitted per DISPLAY/PUNCH operation; if more than one WITH clause is specified, the compiler applies only the options specified in the last one.

Examples

Including All Display Options Except One

This example produces a display of all options, except the journal buffer's history:

```
display journal buffer idmsdmcl.jrnl_buffer
        with all
        without history;
```

DISPLAY/PUNCH ALL Statement

The DISPLAY/PUNCH ALL statement displays all occurrences of a physical database entity.

Syntax

```
►►─┬─ DISplay ─┬─┬─ ALL ─────────────────────────────────┬─ entity-type ─────────►
   └─ PUNch ───┘ └─┬─ FIRst ─┬─┬─────────────────────┬───┘
                   └─ LASt ──┘ ├─ 1 ◄────────────────┤
                               └─ entity-count ──────┘

►─┬────────────────────────────────────┬──────────────────────────────────────►
  └─ WHEre conditional-expression ─────┘

►─┬────────────────────────────┬──────────────────────────────────────────────►
  └─ VERB ─┬─ DISplay ◄──────┬─┘
           ├─ PUNch ─────────┤
           ├─ CREate ────────┤
           ├─ ALter ─────────┤
           └─ DROp ──────────┘

►─┬─────────────────────────────────────────────┬─────────────────────────────►◄
  └─ AS ─┬─ COMments ─┬─┬──────────────────┬────┘
         └─ SYNtax ───┘ └─ RECursive ──────┘
```

Expansion of *conditional-expression*

```
►►─┬─ mask-comparison ─────────────────────────────────┬──────────────────────►
   ├─ value-comparison ────────────────────────────────┤
   └─┬─────────┬─ ( ─┬─ mask-comparison ───┬─ ) ───────┘
     └─ NOT ───┘     └─ value-comparison ──┘

►─┬─────────────────────────────────────────────────────────────────────┬─────►◄
  │ ┌─────────────────────────────────────────────────────────────────┐ │
  └─▼─┬─ AND ─┬─┬─ mask-comparison ───────────────────────────────────┬┴─┘
      └─ OR ──┘ ├─ value-comparison ──────────────────────────────────┤
                └─┬─────────┬─ ( ─┬─ mask-comparison ───┬─ ) ─────────┘
                  └─ NOT ───┘     └─ value-comparison ──┘
```

Expansion of *mask-comparison*

```
►►── entity-option-keyword ───────────────────────────────────────────────────►

►─┬─ CONTAINs ─┬─ 'mask-value' ───────────────────────────────────────────────►◄
  └─ MATCHES ──┘
```

Expansion of *value-comparison*

```
►►─┬─ 'character-string-literal' ─┬───────────────────────────────────────────►
   ├─ numeric-literal ────────────┤
   └─ entity-option-keyword ──────┘

►─┬─ IS ─┬─────────┬─────┬─┬─ 'character-string-literal' ─┬───────────────────►◄
  │      └─ NOT ───┘     │ ├─ numeric-literal ────────────┤
  ├─ NE ─────────────────┤ └─ entity-option-keyword ──────┘
  └─┬─────────┬─┬─┬─ EQ ─┬─┤
    └─ NOT ───┘ │ └─ = ──┘ │
                ├─┬─ GT ─┬─┤
                │ └─ > ──┘ │
                ├─┬─ LT ─┬─┤
                │ └─ < ──┘ │
                ├─ GE ─────┤
                └─ LE ─────┘
```

Parameters

ALL

Lists all occurrences of the requested entity type that the current user is authorized to display.

Online users: With a large number of entity occurrences, ALL may slow response time.

FIRst

Lists the first occurrence of the named entity type.

LASt

Lists the last occurrence of the named entity type.

entity-count

Specifies the number of occurrences of the named entity type to list. 1 is the default.

entity-type

Identifies the entity type that is the object of the DISPLAY/PUNCH ALL request. Valid physical database entity-type values appear in the table under "Usage" below.

WHEre *conditional-expression*

Specifies criteria to be used by the compiler in selecting occurrences of the requested entity type.

The outcome of a test for the condition determines which occurrences of the named entity type the schema or subschema compiler selects for display.

mask-comparison

Compares an entity type operand with a mask value.

entity-option-keyword

Identifies the left operand as a syntax option associated with the named entity type. The table located in the "Usage" section lists valid options for each entity type.

CONTAINs

Searches the left operand for an occurrence of the right operand. The length of the right operand must be less than or equal to the length of the left operand. If the right operand is not contained entirely in the left operand, the outcome of the condition is false.

MATCHES

Compares the left operand with the right operand one character at a time, beginning with the leftmost character in each operand. When a character in the left operand does not match a character in the right operand, the outcome of the condition is false.

'*mask-value*'

Identifies the right operand as a character string; the specified value must be enclosed in quotation marks. *Mask-value* can contain the following special characters:

Special Character	Description
@	Matches any alphabetic character in *entity-option-keyword*.
#	Matches any numeric character in *entity-option-keyword*.
*	Matches any character in *entity-option-keyword*.

value-comparison

Compares values contained in the left and right operands based on the specified comparison operator.

'*character-string-literal*'

Identifies a character string enclosed in quotes.

numeric-literal

Identifies a numeric value.

entity-option-keyword

Identifies a syntax option associated with the named entity type; valid options for each entity type are listed in the table presented under "Usage" below.

IS

Specifies that the left operand must equal the right operand for the condition to be true.

NE

Specifies that the left operand must *not* equal the right operand for the condition to be true.

EQ/=

Specifies that the left operand must equal the right operand for the condition to be true.

GT/>

Specifies that the left operand must be greater than the right operand for the condition to be true.

LT/<

Specifies that the left operand must be less than the right operand for the condition to be true.

GE

Specifies that the left operand must be greater than or equal to the right operand for the condition to be true.

LE

Specifies that the left operand must be less than or equal to the right operand for the condition to be true.

NOT

Specifies that the opposite of the condition fulfills the test requirements. If NOT is specified, the condition must be enclosed in parentheses.

AND

Indicates the expression is true only if the outcome of both test conditions is true.

OR

Indicates the expression is true if the outcome of either one or both test conditions is true.

RECursive

Appends "AS SYNTAX." or "AS COMMENT." to each generated line of output.

Note: For descriptions of the remaining DISPLAY parameters, see Generic DISPLAY/PUNCH Statement (see page 114).

Usage

Output Contains Only Enough Information to Display/Punch Entity

Output produced by DISPLAY or PUNCH ALL consists only of the information necessary to execute a DISPLAY/PUNCH request for each entity occurrence. For example, DMCL occurrences are displayed with their name, and AREA occurrences with their fully qualified name (that is, segmentname.areaname). In an online session, the user can execute the displayed statements by pressing Enter. This two-step process allows the user to scan the names of entity occurrences before submitting the generated statements for execution.

Valid Entity Option Keywords for Conditional Expressions

The following table lists entity type options that you can specify in a conditional expression.

Entity Type	Option
ARCHIVE	FULl <entity-type> NAMe
JOURNALS	<entity-type> JOUrnal name
DISK JOURNALS	NAMe
TAPE JOURNALS	DMCl name
	DDName
	ACCess method
	DATASPACE
	PREpared by
	CREated by
	REVised by
	LAST UPDated by
	DATE LASt CRItical CHAnge
	MONth LASt CRItical CHAnge
	DAY LASt CRItical CHAnge
	YEAr LASt CRItical CHAnge
	DATE last UPDated
	MONth last UPDated
	DAY last UPDated
	YEAr last UPDated
	DATE CREated
	MONth CREated
	DAY CREated
	YEAr CREated

Entity Type	Option
JOURNAL	FULl <entity-type> NAMe
BUFFERS	journal BUFfer name
BUFFERS	NAMe
	PREpared by
	CREated by
	REVised by
	LAST UPDated by
	DATE LASt CRItical CHAnge
	MONth LASt CRItical CHAnge
	DAY LASt CRItical CHAnge
	YEAr LASt CRItical CHAnge
	DATE last UPDated
	MONth last UPDated
	DAY last UPDated
	YEAr last UPDated
	DATE CREated
	MONth CREated
	DAY CREated
	YEAr CREated

Entity Type	Option
DBNAMES	FULl <entity-type> NAMe
DBGROUPS	DBName <entity-type>
	NAMe
	DBTable name
	PREpared by
	CREated by
	REVised by
	LAST UPDated by
	DATE LASt CRItical CHAnge
	MONth LASt CRItical CHAnge
	DAY LASt CRItical CHAnge
	YEAr LASt CRItical CHAnge
	DATE last UPDated
	MONth last UPDated
	DAY last UPDated
	YEAr last UPDated
	DATE CREated
	MONth CREated
	DAY CREated
	YEAr CREated

Entity Type	Option
DBTABLES	DBTable name
	NAMe
	CV system
	SYStem
	PREpared by
	CREated by
	REVised by
	LAST UPDated by
	DATE LASt CRItical CHAnge
	MONth LASt CRItical CHAnge
	DAY LASt CRItical CHAnge
	YEAr LASt CRItical CHAnge
	DATE last UPDated
	MONth last UPDated
	DAY last UPDated
	YEAr last UPDated
	DATE CREated
	MONth CREated
	DAY CREated
	YEAr CREated

Entity Type	Option
DMCLs	DMCl name
	NAMe
	DBTable name
	PREpared by
	CREated by
	REVised by
	LAST UPDated by
	DATE LASt CRItical CHAnge
	MONth LASt CRItical CHAnge
	DAY LASt CRItical CHAnge
	YEAr LASt CRItical CHAnge
	DATE last UPDated
	MONth last UPDated
	DAY last UPDated
	YEAr last UPDated
	DATE CREated
	MONth CREated
	DAY CREated
	YEAr CREated

Entity Type	Option
FILES	FULl file NAMe
	FILe name
	file NAMe
	SEGment name
	DDName
	DSName
	ACCess method
	z/VM USEr id
	z/VM virtual ADDress
	SET name
	PREpared by
	CREated by
	REVised by
	LAST UPDated by
	DATE LASt CRItical CHAnge
	MONth LASt CRItical CHAnge
	DAY LASt CRItical CHAnge
	YEAr LASt CRItical CHAnge
	DATE last UPDated
	MONth last UPDated
	DAY last UPDated
	YEAr last UPDated
	DATE CREated
	MONth CREated
	DAY CREated
	YEAr CREated

Entity Type	Option
PHYSICAL	FULl physical area NAMe
AREAS	physical AREa name
	NAMe
	SEGment name
	PAGe GROup
	area TYPe
	PREpared by
	CREated by
	REVised by
	LAST UPDated by
	DATE LASt CRItical CHAnge
	MONth LASt CRItical CHAnge
	DAY LASt CRItical CHAnge
	YEAr LASt CRItical CHAnge
	DATE last UPDated
	MONth last UPDated
	DAY last UPDated
	YEAr last UPDated
	DATE CREated
	MONth CREated
	DAY CREated
	YEAr CREated

Entity Type	Option
SEGMENTS	SEGment name
	NAMe
	PAGe GROup
	segment TYPe
	PREpared by
	CREated by
	REVised by
	LAST UPDated by
	DATE LASt CRItical CHAnge
	MONth LASt CRItical CHAnge
	DAY LASt CRItical CHAnge
	YEAr LASt CRItical CHAnge
	DATE last UPDated
	MONth last UPDated
	DAY last UPDated
	YEAr last UPDated
	DATE CREated
	MONth CREated
	DAY CREated
	YEAr CREated

Default Order of Precedence Applied to Logical Operators

Conditional expressions can contain a single condition, or two or more conditions combined with the logical operators AND or OR. The logical operator NOT specifies the opposite of the condition. The compiler evaluates operators in a conditional expression 1 at a time, from left to right, in order of precedence. The default order of precedence is as follows:

- MATCHES or CONTAINS keywords
- EQ, NE, GT, LT, GE, LE operators
- NOT
- AND
- OR

If parentheses are used to override the default order of precedence, the compiler evaluates the expression within the innermost parentheses first.

Date Selection Criteria

Date selection in these WHERE clause options:

- DATE CREATED
- DATE LAST UPDATED
- DATE LAST CRITICAL CHANGE

may be specified as a value-comparison string in the form 'MM/DD/YY' or 'CCYY-MM-DD' in the right-hand side of the conditional expression and will be interpreted by the extraction in CCMMDDYY form to accurately determine the relationship of dates. For example, these DISPLAY ALL statements:

```
DISPLAY ALL SEGMENTS
        WHERE DATE CREATED > '01/01/96';

DISPLAY ALL DMCLS
        WHERE DATE LAST CRITICAL CHANGE < '1996-07-14';
```

establishes a search criteria to identify the occurrences whose date values (which are also evaluated in CCYYMMDD form) meet the requirements of the specified string. The DISPLAY ALL process determines that the date '01/01/96' is greater than the date '12/31/95'.

Alternately, you may specify the value-comparison string on either side of the conditional expression in the form 'CCYYMMDD' to achieve the same results.

You can substitute day, month, or year for each of these WHERE clause options. For example, this DISPLAY ALL statement specifies a search condition which is based on month and year.

```
DISPLAY ALL AREAS WHERE MONTH CREATED = '01'
                    AND YEAR  CREATED > '95';
```

Example

The following example displays all AREAS created since June 1, 1986:

```
display all AREAS where date created > '1986-06-01'
    as syntax;
```

ARCHIVE JOURNAL Statements

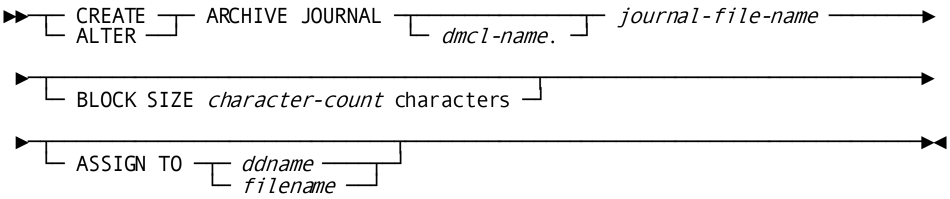
The ARCHIVE JOURNAL statements create, alter, drop, display, or punch the definition of an archive journal file in the dictionary.

Authorization

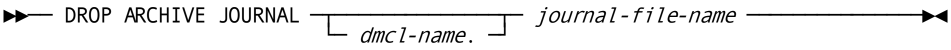
- To create, alter, or drop an archive journal, you must have the following privileges:
  - DBADMIN on the dictionary in which the archive journal definition resides
  - ALTER on the DMCL with which the archive journal is associated
- To display or punch the archive journal, you must have the DISPLAY privilege on the DMCL with which the archive journal is associated or DBADMIN on the dictionary in which the archive journal definition resides.

Syntax

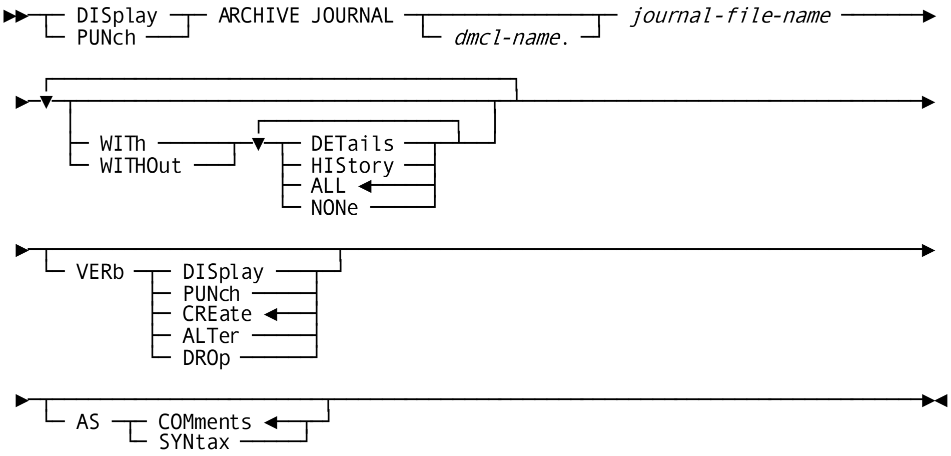
CREATE/ALTER ARCHIVE JOURNAL



DROP ARCHIVE JOURNAL



DISPLAY/PUNCH ARCHIVE JOURNAL



Parameters

dmcl-name

Identifies the DMCL with which the archive journal file is associated. *Dmcl-name* must name an existing DMCL defined to the dictionary. If you don't specify a DMCL name, you must first establish a current DMCL as described in Entity 7.3.3, “Entity Currency" earlier in this chapter.

journal-file-name

Specifies the name of the archive file. *Journal-file-name* must be a 1- through 18-character name that follows the conventions described in 7.3, "Naming Conventions".

Journal-file-name must be unique within the DMCL.

BLOCK SIZE *character-count*

Specifies the number of characters in each block of the archive journal file. This clause is required on a CREATE statement.

The value of *character-count* depends on the operating system:

Operating System	Block Size Range (in bytes)	Comments
z/OS	512 - 327641	Must be greater than or equal to the journal buffer page size and should be sized for efficient tape file storage and access.
z/VSE	512 - 32764	Same as for z/OS.
z/VM	4096	

Note: Maximum for an IBM 3380 device is 32760.

ASSIGN TO

Associates the archive journal file with an external file name. This clause is required on a CREATE statement. The external file name must be unique within the DMCL.

ddname

Specifies the external name for the file under z/OS or z/VM. *ddname* must be a 1- through 8-character value that follows operating system conventions for ddnames.

filename

Specifies the external name for the file under z/VSE. *Filename* must be a 1- through 7-character value that follows operating system conventions for filenames.

DETails

Displays or punches details about the archive journal.

HIStory

Displays or punches:

- The user who defined the archive journal
- The user who last updated the archive journal
- The date the archive journal was created
- The date the archive journal was last updated

ALL

Displays or punches all information about the archive journal. ALL is the default action for a DISPLAY or PUNCH verb.

NONe

Displays or punches the name of the archive journal.

Usage

Archive Journal File Requirement

You must define an archive journal if you are journaling to disk files. When a disk journal file is full, you offload the disk journal to the archive journal. While the offload occurs, CA IDMS/DB journals to another disk journal.

Using Multiple Archive Journals as Backup

You can define multiple archive journals associated with one DMCL. When you invoke the ARCHIVE JOURNAL utility statement to offload a disk journal file, CA IDMS/DB writes the contents of the disk journal to each archive file associated with the DMCL. Therefore, if during the course of manual recovery, an archive file is unreadable, you can attempt recovery using an alternate archive journal file.

Incompatibility of Tape and Archive Journal Files

You cannot include the definition of a tape journal file in the DMCL if you include the definition of disk and archive journal files.

Archive Journal Block Size

When a DMCL is generated, the block size associated with an archive journal is checked to ensure it is not less than the block size of the disk journals. Since the block size of the disk journals is derived from the page size of the journal buffer, if the archive journal's block size is less than the page size of the journal buffer, the page size of the journal buffer is used and a warning message issued.

Examples

Defining an Archive Journal File

The following CREATE ARCHIVE JOURNAL statement defines the archive journal file SYSJRNL:

```
create archive journal idmsdmcl.sysjrnl
   block size 19068 characters
   assign to sysjrnl;
```

Changing the Block Size

The following ALTER ARCHIVE JOURNAL statement changes the block size of the archive journal file SYSJRNL to 32,670 characters:

```
alter archive journal idmsdmcl.sysjrnl
   block size 32670 characters;
```

Dropping an Archive Journal File

The following DROP ARCHIVE JOURNAL statement deletes the definition of the archive journal file SYSJRNL from the dictionary:

```
drop archive journal idmsdmcl.sysjrnl;
```

More Information

- For more information about the procedure for defining disk and archive journals, see the chapter “Defining, Generating, and Punching a DMCL”.
- For more information about journaling procedures, such as offloading, see the chapter “Journaling Procedures".
- For more information about defining disk journal files, see the section “DISK JOURNAL Statements”.

AREA Statements

The AREA statements create, alter, delete, display, or punch the definition of an area in the dictionary.

Authorization

- To create, alter, or drop an area, you must have the following privileges:
  - DBADMIN on the dictionary in which the area definition resides
  - ALTER on the segment with which the area is associated
- To display or punch an area, you must have DISPLAY privilege on the segment with which the area is associated or DBADMIN on the dictionary in which the area definition resides

Syntax

CREATE/ALTER AREA

```
►►─┬─ CREATE ─┬─ physical AREA ─┬─────────────────┬─ area-name ──────────►
   └─ ALTER ──┘                 └─ segment-name. ─┘

 ►─┬─────────────────────────────────────────┬──────────────────────────►
   ├─ initial-page-range-specification ──────┤
   └─ EXTEND SPACE extend-page-count pages ──┘

 ►─┬─────────────────────────────────────┬──────────────────────────────►
   └─ PAGE SIZE character-count characters ─┘

 ►─┬──────────────────────────────────────────────────────────────┬─────►
   └─ PAGE RESERVE size ─┬─ 0 ◄──────────────────────┬─ characters ─┘
                         └─ reserve-character-count ─┘

 ►─┬───────────────────────────────────────────────────────────┬────────►
   └─ ORIGINAL PAGE SIZE original-character-count characters ─┘

 ►─┬─────────────────────────┬──────────────────────────────────────────►
   └─ STAMP BY ─┬─ TABLE ─┬──┘
                └─ AREA ──┘

 ►─┬──────────────────────────────────┬─────────────────────────────────►
   └───── TIMESTAMP timestamp-value ──┘

 ►─┬───────────────────────────────┬────────────────────────────────────►
   │  ┌───────────────────────────┐│
   └──▼─ symbol-specification ────┴┘

 ►─┬────────────────────────────────┬───────────────────────────────────►◄
   │  ┌────────────────────────────┐│
   └──▼───file-specification ──────┴┘
```

DROP AREA

```
►►── DROP physical AREA ─┬─────────────────┬─ area-name ──────────►◄
                         └─ segment-name. ─┘
```

Expansion of *initial-page-range-specification*

```
►►─── PRIMARY SPACE primary-page-count pages FROM page start-page ──────►

 ►─┬─────────────────────────────────────┬──────────────────────────────►◄
   └─ MAXIMUM SPACE max-page-count pages ─┘
```

Expansion of *symbol-specification*

```
►►──┬───────────────┬──────────────────────────────────────────────────────────►
    ├─┬─ ADD ◄────┤
    │ └─ INClude ─┘ │
    └─┬─ DROP ────┤
      └─ EXClude ─┘

 ►─┬─ SUBAREA symbolic-subarea-name ─┬─────────────────────────┬────────────┬─►◄
   │                                 └ subarea-specification ─┘            │
   ├─ SYMBOLIC DISPLACEMENT symbolic-displacement-name ─┬──────────────────┬┤
   │                                                    └ page-cnt pages ─┘│
   └─ SYMBOLIC INDEX symbolic-index-name ─┬─────────────────────────┬──────┘
                                          └ index-specification ─┘
```

Expansion of *subarea-specification*

```
►►─┬─ FROM page start-page THRU page end-page ──────────────────────────────────┬─►◄
   ├─ SPACE subarea-page-count pages FROM page subarea-start-page ─────────────┤
   └─ OFFSET ─┬─ 0 ◄──────────────────────┬─ FOR ─┬─ 100 PERCENT ◄────┤
              ├─ offset-page-count PAGEs ─┤       ├─ percent PERCENT ──┤
              └─ offset-percent PERCENT ──┘       └─ page-count PAGEs ─┘
```

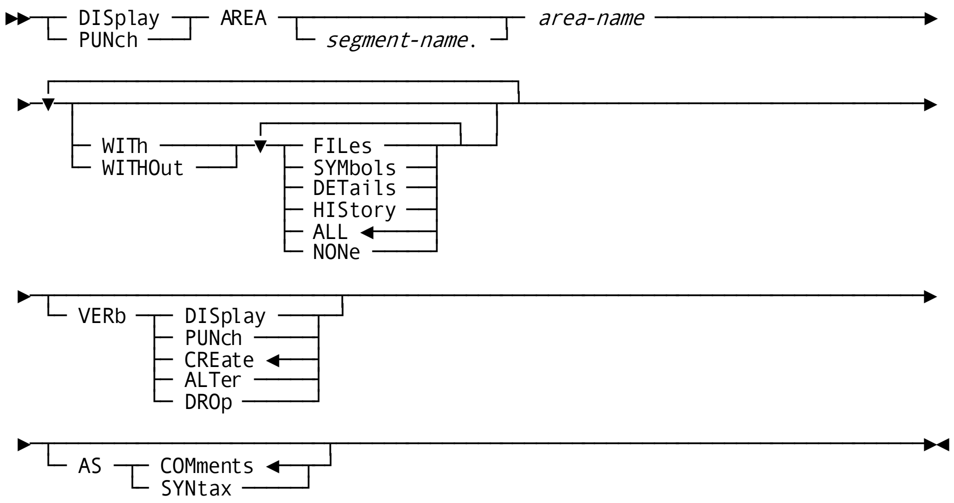
Expansion of *index-specification*

```
►►─┬ BLOCK CONTAINS key-count keys ─┬──────────────────────────────┬──────┬─►◄
   │                                └ DISPLACEMENT page-count pages ┘      │
   └ BASED ON ─┬─────────────┬─ KEY LENGTH key-length ┬───────────────────────┬┘
               ├ SORTED ◄──┤                          └ FOR index-cnt RECORDS─┘
               └ UNSORTED ─┘
```

Expansion of *file-specification*

```
►►──┬───────────────┬──┬─ FILE file-name ─────────────────────────┬───────────►
    ├─┬─ ADD ◄────┤  └─ PATH FILE native-vsam-file-name ─┘
    │ ├─ WITHIN ──┤ │
    │ └─ INClude ─┘ │
    └─┬─ REMOVE ──┤
      ├─ DROP ────┤
      └─ EXClude ─┘

 ►──┬──────────────────────────────────────────────────────────────┬──────────►◄
    └─ FROM start-block ─┬─ THRU end-block ──────────────────┤
                         └─ FOR ─┬─ ALL blocks ──────────┤
                                 └─ block-count blocks ─┘
```

DISPLAY/PUNCH AREA



Parameters

segment-name

Specifies the segment associated with the area. *Segment-name* must identify a segment defined in the dictionary.

If you do not specify a segment name when you issue an AREA statement, you must first establish a current segment as described in 7.3.3, “Entity Currency" earlier in this chapter.

area-name

Specifies the name of the area. *Area-name* must be a 1- through 18-character name that follows the conventions described in 7.3, "Naming Conventions".

Area-name must be unique within the segment associated with the area.

Important! If the area is associated with an SQL segment in an application dictionary, you must drop any tables or indexes associated with the area before you attempt to delete the area by issuing a DROP AREA statement.

Important! If the area is associated with a non-SQL segment, the name of the area must be the same as the area defined in the non-SQL schema.

initial-page-range-specification

Specifies the initial page range assigned to the area. This clause is required on a CREATE statement.

Native VSAM: For special considerations that apply to the page ranges of native VSAM data sets, see the "Usage" topic in this section.

PRIMARY SPACE *primary-page-count*

Specifies the initial number of pages to be included in the area. *Primary-page-count* must be an integer in the range 2 through the maximum number of pages determined by the MAXIMUM RECORDS clause of the SEGMENT statement. The upper limit is 1,073,741,821.

Important! This parameter establishes the default CALC page range of the area and should not be specified with new values on an ALTER AREA request unless the area is empty or is to be reloaded using the RELOAD or REORG utilities.

FROM page *start-page*

Specifies the page number of the first page in the area. *Start-page* must be an integer in the range 1 through the maximum number of pages determined by the MAXIMUM RECORDS clause of the SEGMENT statement. The upper limit is 1,073,741,821.

MAXIMUM SPACE *max-page-count* pages

Specifies the largest number of pages that can be included in the area. *Max-page-count* must be:

- An integer in the range 2 through the maximum number of pages determined by the MAXIMUM RECORDS clause of the SEGMENT statement; the upper limit is 1,073,741,821.
- Greater than or equal to the primary page count for the area

The default maximum number of pages is the area's primary page count.

Native VSAM: If specified, MAXIMUM SPACE must equal the primary page count.

EXTEND SPACE *extend-page-count*

On an ALTER AREA statement, specifies a number of pages to be added to the area. The new pages are numbered starting after the last page currently in the area.

Extend-page-count must be an integer in the range 1 through the maximum number of pages determined by the MAXIMUM RECORDS clause of the SEGMENT statement. The upper limit is 1,073,741,818. The number of new pages plus the number of existing pages cannot exceed the maximum number of pages allowed for the area.

When you add pages to an area, you must also associate the added pages with either:

- One or more additional files
- File blocks beginning at the end of the last file with which the area is associated

Added pages are automatically associated with file blocks, by specifying the 'WITHIN FILE'-clause without the 'FROM'-clause for the <file-name> (if only 1 file is associated with the area) or for the last <file-name> (if more than 1 file is associated with the area). All other changes in the assignment of file blocks require first an EXCLUDE of the <file-name(s)>, followed by a new 'WITHIN FILE <file-name> FROM'-clause.

Important! When specifying an EXTEND SPACE parameter, do not specify a PRIMARY SPACE parameter which alters the original page range of the area.

Native VSAM: Do not specify the EXTEND SPACE clause.

Note: See the Usage section for guidelines about using this parameter.

Note: This parameter is not valid on the CREATE AREA statement.

PAGE SIZE *character-count*

Specifies the number of characters in each page of the area. This clause is required on a CREATE statement. *Character-count* must be a multiple of 4 in the range 48 through 32,764 and must be at least 40 bytes larger than the largest fixed-length record or uncompressed row in the area. Some operating systems may not support a page size of 32764 characters. Check your operating system limitations.

Native VSAM: Do not specify the PAGE SIZE clause.

PAGE RESERVE SIZE *reserve-character-count*

Specifies the number of characters to be reserved on each page to accommodate increases in the length of records or rows stored on the page. Reserved space will be used for:

- SR8 index records, which vary in length at the bottom level of the index. The length of a bottom-level SR8 record can change due to any operation that updates an indexed record. Reserved space is not available for new SR8 records.
- Variable-length records that expand during DML MODIFY operations.
- Compressed rows or records whose physical length increases due to a change in the data values.

Reserve-character-count must be either 0 or:

- A multiple of 4 in the range 48 through 32,716
- Less than or equal to the size of a page in the area minus 48

The default is 0.

Native VSAM: Do not specify this clause.

ORIGINAL PAGE SIZE *original-character-count*

Specifies the page size of the area when it was last formatted. This clause must be specified the first time the page size of an area is increased using the EXPAND PAGE utility statement, and should not be specified again unless you reformat the area using the new specification.

Original-character-count must be a multiple of 4 in the range 48 through 32764 and cannot be greater than the value specified for the PAGE SIZE clause. The default on a CREATE AREA statement is the value specified for the PAGE SIZE clause.

Native VSAM: Do not specify this clause.

STAMP BY TABLE

On a CREATE AREA statement, directs CA IDMS/DB to update the synchronization stamp for an individual table in the area when the definition of the table or any associated CALC key, index, or referential constraint is modified. This clause is valid only for areas that are associated with an SQL segment.

STAMP BY TABLE overrides the synchronization stamp specification defined for the segment with which the area is associated.

Note: This parameter is not valid on the ALTER AREA statement.

STAMP BY AREA

On a CREATE AREA statement, directs CA IDMS/DB to maintain a synchronization stamp for the area as a whole in addition to the synchronization stamps for individual tables. CA IDMS/DB updates the stamps for both the individual table and the whole area when the definition of any table in the area or any associated CALC key, index, or referential constraint is modified.

This clause is valid only for areas that are associated with an SQL segment.

STAMP BY AREA overrides the synchronization stamp specification defined for the segment with which the area is associated.

Note: This parameter is not valid on the ALTER AREA statement.

TIMESTAMP *timestamp-value*

Specifies the value of the synchronization stamp to be assigned to the area. *Timestamp-value* must be a valid external representation of a timestamp. This clause is valid only for areas associated with an SQL segment and for which area-level stamping is in effect.

symbol-specification **:pd**

ADD

For areas associated with non-SQL segments, specifies a value for a symbolic parameter defined in a non-SQL schema definition. ADD is the default.

Note: If the symbolic parameter is already defined to the area, CA IDMS/DB *updates* its value.

DROP

For areas associated with non-SQL segments, removes the symbolic parameter.

To drop a symbolic parameter, specify only the name of the symbol to be dropped. Optional clauses, such as *subarea-specification*, are not allowed.

SUBAREA *symbolic-subarea-name*

Names a symbolic parameter that represents a subdivision of the area's page range. *Symbolic-subarea-name* is a 1- to 18-character name that follows the conventions described in 7.3, "Naming Conventions". *Symbolic-subarea-name* must be unique within the subareas defined for the area.

subarea-specification

Specifies an actual page range for the subarea or a relative page range for the subarea based upon the total number of pages defined for the area. If you do not specify an actual or relative page range for the subarea, the default is the page range of the area expressed as this offset specification:

```
offset 0 pages for 100 percent
```

FROM page *start-page*

Specifies the starting page for the subarea. *Start-page* must be an integer in the range 1 through the high page number of the area.

THRU page *end-page*

Specifies the last page for the subarea. *End-page* must be an integer:

- Within the page range defined for the area
- Greater than the value specified for *start-page*

SPACE *subarea-page-count* pages

Specifies the number of pages to be included in the subarea. *Subarea-page-count* is an integer in the range 1 through the number of pages in the area.

FROM page *subarea-start-page*

Specifies the first page of the subarea. *Subarea-start-page* must be an integer in the range 1 through the high page number of the area.

OFFSET

Specifies a relative page range for the subarea, in terms of either a percentage of the area or a displacement relative to the first page of the area. The assigned relative page range must fall within the page range for the area.

***offset-page-count* PAGEs**

Determines the first page of the subarea within the area. CA IDMS/DB uses the calculation below to determine the relative page number:

```
first subarea page = (LPN + offset-page-count)

     where LPN = the lowest page number in the area
```

Offset-page-count must be an integer in the range 0 through the number of pages in the area minus 1.

***offset-percent* PERcent**

Determines the first page of the subarea within the area based on the lowest page number of the area and the total number of pages in the area:

```
first subarea page = (LPN + (PPC * offset-percent * .01))

     where LPN = the lowest page number in the area
       and PPC = the primary page count
```

Offset-percent must be an integer in the range 0 through 100.

FOR *page-count* PAGEs

Determines the last page of the subarea within the area based on the first page of the subarea:

```
last subarea page = (FSP + page-count - 1)

     where FSP = the first subarea page
                 (determined by calculations above)
```

The calculated page must not exceed the highest page number in the area.

FOR *percent* PERcent

Determines the last page of the subarea within the area based on the first page of the subarea and the total number of pages in the area:

```
last subarea page = (FSP + (TNP * percent * .01) - 1)

     where FSP = the first page in the subarea
                 (determined by calculations above)
       and TNP = the total number of pages in the area
```

Percent must be an integer in the range 1 through 100. The default is 100. If *percent* causes the calculated last page of the subarea to be greater than the highest page number in the area, the compiler ignores the excessive page numbers, and CA IDMS/DB will store the record occurrences up to and including the last page in the area.

SYMBOLIC DISPLACEMENT *symbolic-displacement-name*

Names a symbolic parameter that represents the displacement of member records that participate in a VIA set from the owner record of the set. *Symbolic-displacement-name* is a 1- to 18-character name that follows the conventions described in 7.3, "Naming Conventions". *Symbolic-displacement-name* must be unique within the symbolic displacement names defined to the area.

page-cnt-pages

Specifies how many pages separate the member record of a VIA set from the owner record. *Page-cnt-pages* is an integer in the range 0- through 32767.

SYMBOLIC INDEX *symbolic-index-name*

Names a symbolic parameter that represents index characteristics. *Symbolic-index-name* is a 1- to 18-character name that follows the conventions described in 7.3, "Naming Conventions". *Symbolic-index-name* must be unique within the symbolic index names defined to the area.

index-specification

Specifies either:

- The values that represent the number of entries in an SR8 record and the displacement of bottom-level SR8 records from the remainder of the index.
- The values that are used to calculate the number of SR8 entries and the displacement.

BLOCK CONTAINS *key-count* keys

Specifies the maximum number of entries in each internal index record (SR8 system record). *Key-count* must be an integer in the range 3 through 8180.

DISPLACEMENT *page-count* pages

Indicates the number of pages bottom-level SR8 records are displaced from the top of the index. *Page-count* must be either 0 or an integer in the range 3 through 32,767. The default is 0, which means bottom-level index records are not displaced.

BASED ON KEY LENGTH *key-length*

Calculates the size of the index block and displacement based upon the length of the key fields and the number of entries in the index. Specify *key-length* as:

- 0, for unsorted indexes
- 0, for indexes sorted by db-key
- An integer in the range 1 through 256 for other indexes

SORTED

Indicates that the index keys are sorted.

UNSORTED

Indicates that the index keys are not sorted.

FOR *index-cnt* RECORDS

Specifies an estimated number of record occurrences to be indexed. *Index-cnt* is an integer in the range 0 through 2,147,483,647. The default is 1000. See "Usage" for further information.

file-specification

Specifies the file(s) to which pages in the area map. An area can map to one or more files.

ADD FILE *file-name*

Associates the area with the named database file or native VSAM file that has an access method of KSDS, ESDS or RRDS. *File-name* must identify a file that:

- Is associated with the same segment as the area
- Is not defined with PATH as an access method

You can associate an area with 1 through 32,767 files. Pages in the area are mapped consecutively to blocks in the first file named, then to blocks in the second file named, and so on. If any files are associated with the area, you must identify enough file blocks to accommodate all the pages in the area. **Native VSAM**: Native VSAM files with access method KSDS, ESDS, or RRDS must map to one and only one area. Likewise, the area must map to one and only one native VSAM file and PATH file.

DROP FILE *file-name*

Dissociates the area from the named file. *File-name* must identify a database file previously associated with the area.

If you dissociate a file from an area, you must identify enough additional file blocks in the same ALTER AREA statement to accommodate the pages that no longer map to the file, unless all files are dissociated from the area.

PATH FILE *native-vsam-file-name*

Identifies a native-VSAM PATH file for the area. *Native-vsam-file-name* is a 1- to 18-character name of a PATH file defined to the segment. The following restrictions apply:

- The access method defined for *native-vsam-file-name* on a CREATE/ALTER FILE statement must be PATH.
- The file cannot map to any other areas
- The area must map to a file whose access method is KSDS or ESDS

FROM *start-block*

Specifies the number of the first block in the named file to be associated with the area. *Start-block* must be an integer in the range 1 through 2,147,483,646. The default depends on the verb:

- For CREATE AREA, the default is 1
- For ALTER AREA with*out* the EXTEND SPACE clause, the default is 1
- For ALTER AREA with the EXTEND SPACE clause, the default is the current high block number of the file plus 1

THRU *end-block*

Specifies the number of the last block in the named file to be associated with the area. *End-block* must be an integer in the range 2 through 2,147,483,647.

FOR ALL

Specifies that blocks in the named file are to be associated with the area for the entire page range of the area, or, if specified for an ALTER AREA with an EXTEND SPACE clause, for the number of pages in the extended space.

FOR *block-count* blocks

Specifies the number of blocks in the named file to be associated with the area. *Block-count* must be an integer in the range 2 through 2,147,483,647.

FILes

Displays or punches information about all files to which the area is mapped.

SYMbols

Displays or punches information about all symbols defined to the area.

DETails

Displays or punches details about the area.

HIStory

Displays or punches:

- The user who defined the area
- The user who last updated the area
- The date the area was created
- The date the area was last updated

ALL

Displays or punches all information about the area. ALL is the default action for a DISPLAY or PUNCH verb.

NONe

Displays or punches the name of the area.

Usage

Unique Page Range

The range of pages reserved for an area is defined by the FROM PAGE parameter in conjunction with the MAXIMUM SPACE parameter (or the PRIMARY SPACE parameter if you do not specify MAXIMUM SPACE). This page range must not overlap the page range for:

- Any other area contained in the segment
- Any other area in a DMCL in which the area's segment is included if the page groups are the same

Contiguity of Page Ranges

Page ranges within a segment can be, but do not have to be, contiguous with one another.

Page Range Limits Depend on Maximum Number of Records Per Page

The highest page number for an area depends on the maximum number of records or rows that can fit on a single page. Use the table provided under "Usage" in 7.16, “SEGMENT Statements” to determine the highest page number.

Page Ranges for CALC Records

The last page of a subarea that can be used to store CALC record occurrences depends on the type of offset specification:

- For page offsets, the last page of the CALC range is the last page of the subarea.
- For percentage offsets, this calculation is used to determine the last page of the CALC range:

  ```
  calc-lastpage-of-subarea =
      firstpage-of-subarea + percent * primary-page-count * .01
  ```

What Happens to Offsets When You Expand an Area

When you expand an area by using the EXTEND SPACE clause on the ALTER AREA command, the following occurs to the first page, last page, and CALC last page of a subarea:

- The first page does not change
- The last page changes if you specified a percentage offset; CA IDMS/DB allows CALC overflow records and records with other location modes to be stored in the expanded space
- The last page of the CALC range does not change; that is, CALC records continue to target to the original page range assigned to the subarea

Note: You must exercise care when expanding an area containing subarea definitions that use offset percentages because subareas can overlap after the EXTEND SPACE is performed. For example:

Given the following subarea allocations for an area containg 1000 pages, the page ranges are as follows:

Definition	LoPage	HiPage
SUB1 OFFSET 0% FOR 25%	1	250
SUB2 OFFSET 25% FOR 25%	251	500
SUB3 OFFSET 50% FOR 25%	501	750
SUB4 OFFSET 75% FOR 25%	751	1000

If an EXTEND SPACE is executed, and 1000 more pages are added to the area, the allocations are as follows:

Sub Area	LoPage	HiPage
SUB1	1	500
SUB2	251	750
SUB3	501	1000
SUB4	751	1250

You can see that adding 1000 pages to the area did not significantly increase the space available in which to store records, nor did the additional space get used by any areas mapped to defined subarea definitions—only SUB4 benefits by the additional pages.

Percentage Offsets Most Flexible

Percentage offset specifications are the most flexible in terms of database maintenance. As the database grows and must eventually be expanded, the areas of the database must also be expanded. If you use percentage offsets, CA IDMS/DB automatically assigns record occurrences to the appropriate percentage of the new area.

Page Range for RRDS Native VSAM Areas

CA IDMS/DB constructs the db-key for a record in an RRDS native VSAM area in the following manner:

```
dbkey = low-dbkey-of-area + relative-record-number
```

Therefore, for an RRDS file, the number of pages specified by the page range must be calculated as follows (rounded up to the next integer):

```
number-of-pages =
 (number-of-vsam-records-in-file + 1) / (maximum-records-per-page + 1)
```

Note: Maximum-records-per-page is specified on the CREATE SEGMENT statement and determines the format of the database keys for records in areas that are contained in the segment.

Page Range for RRDS Native VSAM Areas

CA IDMS/DB constructs the db-key for a record in a KSDS native VSAM area by randomizing the record's prime key to a database key in the database key range for the area. Therefore, for a KSDS file, a rule-of-thumb for calculating the page range is as follows (rounded up to the next integer):

```
number-of-pages = number-of-vsam-records-in-file / x

where x =  10 if number-of-vsam-records-in-file < 100,000
           100 if number-of-vsam-records-in-file > 100,000
```

The idea is to specify a page range that minimizes the probability of constructing duplicate keys without specifying an excessive number of pages for the area.

Page Range for ESDS Native VSAM Areas

CA IDMS/DB constructs the db-key for a record in an ESDS native VSAM area in the following manner:

```
dbkey = low-dbkey-of-area + relative-byte-address
```

Therefore, for an ESDS file, the number of pages specified by the page range must be calculated as follows (rounded up to the next integer):

```
number-of-pages = total-bytes-in-file / (maximum-records-per-page + 1)
```

Note: Maximum-records-per-page is specified on the CREATE SEGMENT statement and determines the format of the database keys for records in areas that are contained in the segment.

Physical Device Blocking

A database page is a fixed block. As a general rule, you should use pages that are an even fraction of the track size.

The following table lists the optimal page sizes by device type for five IBM disk drives. Manufacturers of other brands of direct access storage devices (DASD) should be able to provide similar information for their own equipment.

Per track	3330	3340	3350	3375	3380	3390
1	13028	8368	19068	32764	32764	32764
2	6444	4100	9440	17600	23476	27996
3	4252	2676	6232	11616	15476	18452
4	3156	1964	4628	8608	11476	13680

Per track	3330	3340	3350	3375	3380	3390
5	2496	1540	3664	6816	9076	10796
6	2056	1252	3020	5600	7476	8904
7	1744	1052	2564	4736	6356	7548
8	1508	896	2220	4096	5492	6516
9	1324	780	1952	3616	4820	5724
10	1180	684	1740	3200	4276	5064
11	1060	608	1564	2880	3860	4564
12	960	544	1416	2592	3476	4136
13	876	488	1296	2368	3188	3768
14	804	440	1180	2176	2932	3440
15	740	400	1096	2016	2676	3172

Note: The bytes per page for FBA devices must be a multiple of 512.

Note: On z/VM, the size of a database page must be less than or equal to 4096 bytes.

Note: For VSAM database files the character-count must be at least 8 bytes larger than the page size.

Synchronization Stamps

If you expect frequent changes to the definitions of SQL tables, you should maintain synchronization stamps at the table level. If you do not expect frequent changes, you should maintain stamps at the area level.

Specifying a Synchronization Stamp Value

When defining or altering an area for which area-level stamping is in effect, you can specify an explicit value for its synchronization stamp. This allows you to create databases that have identical physical attributes and can therefore be accessed through a single schema definition.

Since an area's synchronization stamp is updated each time any DDL statement affecting the area is issued, the synchronization stamp must be set after issuing the SQL DDL statements that define the database.

Care should be exercised when specifying a specific timestamp, since its purpose is to enable the detection of discrepancies between an entity and its definition. If explicitly specified, the timestamp should always be set to a new value following a definitional change so that the change is detectable to the run time system.

Contiguity of File Blocks

Block ranges within a file associated with more than one area must be contiguous.

To specify that all pages of the area map to all pages of the file, specify:

```
...from 1 for all
```

on the file specification.

If the file has multiple areas associated with it, the block range will overlap if both of the areas map to the file having this specification. You can map the first area using "FROM 1 FOR ALL", but you must map the second area "FROM *last-block-of-the-file*+1 FOR ALL".

Device Types or Access Methods May Limit the Number of File Blocks

Device types or access methods may further restrict the number of blocks allowed in a file. For example, a maximum of 65,535 tracks can be addressed in BDAM files.

Note: For more information about device types and access methods, see Allocating and Formatting Files.

Native VSAM File Restrictions

An area that maps to native VSAM files has the following restrictions:

- A native VSAM file defined with an access method RRDS, KSDS, or ESDS can map to one and only one area
- An area that maps to a native VSAM file must map to one and only one file
- If an area is associated with one or more files defined with PATH as an access method, then:
  - The area must map to either an ESDS or KSDS file
  - The PATH file must not be associated with any other area

Index Calculations

The following algorithms are used to calculate BLOCK CONTAINS *key-count* and the DISPLACEMENT *page-count* values for symbolic index parameters when the BASED ON clause is specified.

Index block:

```
Step 1:  Assuming 3 SR8's per page, compute the following:

         The maximum size of the variable portion of an SR8:
            ( (Page size - Page reserve - 32) / 3 ) - 40 = SR8-vsize

         The maximum number of entries in an SR8:
                Sorted index:  (SR8-vsize / ( 8 + Keylen) ) - 2
              Unsorted index:  (SR8-vsize / 4 ) - 1

         If the number of SR8 entries is less than 3, set it to 3; if
         greater than 8180, set it to 8180.

Step 2:  Establish the number of index entries:  Use the FOR index-cnt
         value, if specified, or 1000.
```

Step 3: Estimate the number of entries per SR8 for a 3-level index: Find the first entry in the following table whose Number of Entries column is greater than or equal to the value established in Step 2.

Number of entries	Number of SR8 entries
1,000	10
15,625	25
125,000	50
512,000	80
1,000,000	100
2,000,376	126
3,375,000	150
5,359,375	175
8,000,000	200
15,625,000	250
-1	8180

Step 4: Determine the INDEX BLOCK value: Use the lesser of the Number of SR8 entries from the table and the value from Step 1 as the INDEX BLOCK (IBC) value in the remaining calculations.

Displacement:

For unsorted indexes, the displacement is set to 0; for sorted indexes, it is calculated as follows:

Step 1: Calculate the number of bottom level and higher level SR8s:

```
      Set N              = #-of-entries
      High-level-SR8s    = 0
      Bottom-level-SR8s  = 1

      Repeat

        N = (N + IBC - 1) / IBC (truncate)

        If N = 1, exit

        If High-level-SR8s = 0,
              High-level-SR8s    = 1
              Bottom-level-SR8s  = N
        Else High-level-SR8s = High-level-SR8s + N

Set Total-SR8s = High-level-SR8s + Bottom-level-SR8s
```

```
Step 2:   Determine the number of SR8s per page:

                Calculate size of an SR8:

                        SR8-size = 32 + (IBC + 1) * (keylen + 8)

                Calculate number of SR8s per page:

                        (Page-size - Page-reserve - 32) / (SR8-size + 8)

Step 3:   Establish the INDEX DISPLACEMENT:

                If Number of Higher Level SR8s is less than 2, set
                the DISPLACEMENT = High-level-SR8s.  (For a one or
                two-level index, displacement will be 0 or 1
                respectively.)

                If Number of Higher Level SR8s is greater than 1,
                compute the displacement:

                  (High-level-SR8s + SR8s-per-page - 1)
                  -------------------------------------  + 1 (truncate)
                            SR8s-per-page

                If the calculate displacement is greater than the number
                of pages in the area containing the index, then:

                  Displacement = Number of pages in area / 2
```

Examples

Mapping to a Single File

The CREATE AREA statement below defines an area that has only one associated file. All 100 pages in the area will map to the first available 100 blocks in the file.

```
create area demoseg.emp_space
   primary space 100 pages
   page size 4276
   within file demoseg.emp_file;
```

Mapping to Two Files

The CREATE AREA statement below defines an area that maps to two files. The first 500 pages in the area map to the first 500 blocks in the PUB_FILE_1 file. The second 500 pages in the area map to 500 blocks of the PUB_FILE_2 file, starting at block number 1001.

```
create area salesseg.sales_space
   primary space 1000 pages
   from page 85001
   maximum space 1500 pages
   page size 3820 characters
   page reserve size 800 characters
   within file pub_file_1
      from 1 for 500
   within file pub_file_2
      from 1 for 500;
```

Adding Pages to an Area

The ALTER AREA statement below adds 200 pages to the SALES_SPACE area. The new pages are mapped to the PUB_FILE_3 file.

```
alter area salesseg.sales_space
   extend space 200 pages
   within file pub_file_3
   from 1 thru 200;
```

Dropping an Area

The following DROP AREA statement deletes the definition of the SALES_SPACE area from the dictionary. If SALESSEG is defined as an SQL segment, then you must first drop all tables and indexes associated with the area:

```
drop area salesseg.sales_space;
```

More Information

- For more information about defining segments, areas, and files, see Chapter 4, “Defining Segments, Files, and Areas".
- For more information about modifying segments, areas, and files, see Chapter 27, “Modifying Physical Database Definitions".

BUFFER Statements

The BUFFER statements create, alter, drop, display, or punch the definition of a database buffer in the dictionary. You must define at least one database buffer for a DMCL.

Authorization

- To create, alter, or drop a database buffer, you must have the following privileges:
  - DBADMIN on the dictionary in which the database buffer definition resides
  - ALTER on the DMCL with which the database buffer is associated
- To display or punch the database buffer, you must have DISPLAY privilege on the DMCL with which the database buffer is associated or DBADMIN on the dictionary in which the buffer definition resides

Syntax

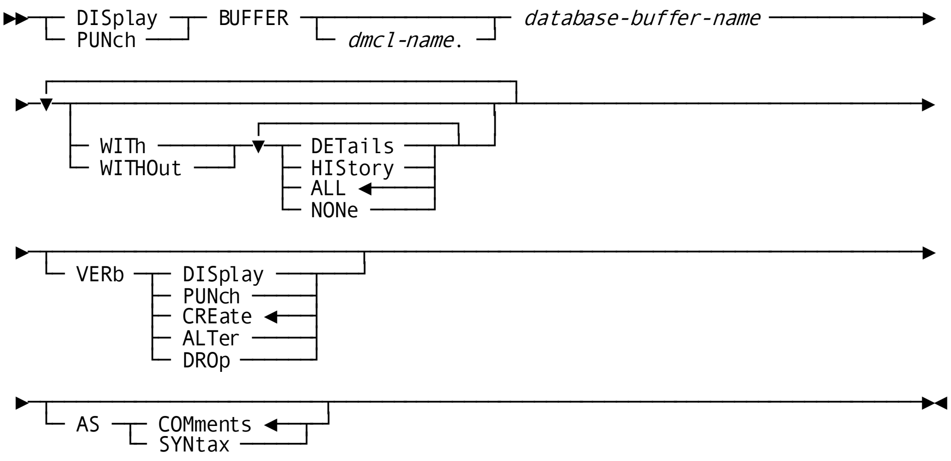
CREATE/ALTER BUFFER

```
►►─┬─ CREATE ─┬─ BUFFER ─┬──────────────┬─ database-buffer-name ──────────►
   └─ ALTER ──┘          └─ dmcl-name. ─┘

 ►─┬───────────────────────────────────────┬─────────────────────────────►
   └─ PAGE SIZE character-count characters ─┘

 ►─┬─────────────────────────────────────────────────────────────────┬───►
   └─ NATIVE VSAM ─┬─ LSR KEYLEN lsr-key-length ─┬─ STRNO string-number ─┘
                   └─ NSR BUFNI nsr-buffer-count ─┘

 ►─┬──────────────────────────────────────────────────────────────────┬──►
   └─ LOCAL MODE BUFFER PAGES local-mode-page-count ─┬─────────────────┬┘
                                                     ├─ OPSYS storage ◄┤
                                                     └─ IDMS storage ──┘

 ►─┬────────────────────────────────────────────────────────────┬────────►◄
   └─┬─ CENTRAL VERSION ─┬─ MODE BUFFER ─┬───────────────────┬──┘
     └─ CV ──────────────┘               └─ cv-buffer-options ─┘
```

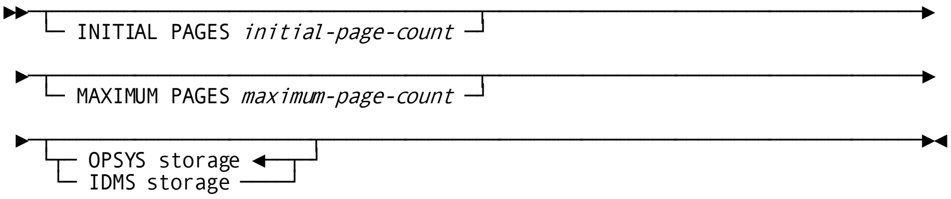
DROP BUFFER

```
►►── DROP BUFFER ─┬──────────────┬─ database-buffer-name ─────────────────►◄
                  └─ dmcl-name. ─┘
```

DISPLAY/PUNCH BUFFER



Expansion of *cv-buffer-options*


INITIAL PAGES initial-page-count
MAXIMUM PAGES maximum-page-count
OPSYS storage
IDMS storage

Parameters

dmcl-name

Identifies the DMCL with which the database buffer is associated. *Dmcl-name* must name an existing DMCL defined to the dictionary. If you don't specify a DMCL name, you must establish a current DMCL as described in 7.3.3, “Entity Currency".

database-buffer-name

Specifies the name of the buffer being created. *Database-buffer-name* must be a 1-through 18-character name that follows the conventions described in 7.3, “Naming Conventions".

Database-buffer-name must be unique among the database and journal buffer names within the DMCL. From 1 to 32,767 database buffers can be defined to a single DMCL.

PAGE SIZE *character-count*

Specifies the number of characters in each page of the buffer. This clause is required on a CREATE statement. The buffer page size determines the size of the largest database page or VSAM control interval that can be written to the buffer.

The value of *character-count* depends on the type of buffer being defined:

File	Buffer Type	Valid Page Sizes (in bytes)
VSAM database file		48 - 32764; multiple of 4 1 2
Native VSAM file3	LSR	512, 1024, 2048, or multiple of 4096 up to 28672
	NSR	512 - 8192; multiple of 512 8193 - 30720; multiple of 2048

Note: For VSAM database files, character-count must be at least 8 bytes larger than the size of the database page.

Note: For native VSAM files, the PAGE SIZE clause must be greater than or equal to the largest control interval of a file that maps to the buffer.

NATIVE VSAM

Specifies a buffer for use with native VSAM data sets.

LSR KEYLEN *lsr-key-length*

Specifies an LSR (local shared resource) buffer. Only one is allowed per DMCL.

Lsr-key-length specifies the maximum key length for all native VSAM files using the buffer, where *lsr-key-length-n* is an integer in the range 1 through 255.

NSR BUFNI *index-buffer-count*

Specifies an NSR (nonshared resource) buffer. Any number of these are allowed.

Index-buffer-count specifies the number of index buffers VSAM uses to transfer the contents of index entries between main memory and auxiliary storage. It is an integer in the range *string-number* through 32767.

STRNO *string-number*

Specifies the maximum number of concurrent requests permitted against all areas associated with files that are assigned to the buffer, where *string-number* is an integer in the range 1 through 255.

LOCAL MODE BUFFER PAGES *local-mode-page-count*

Specifies the number of pages to be included in the buffer when the database is used in local mode. Valid values for *local-mode-page-count* appear as follows:

Buffer Type	Valid Values
Non-native VSAM buffers	3 to 16,777,2141; default 3
Native VSAM buffers	2 to 256; must be greater than the value assigned to STRNO in the NATIVE VSAM clause above

Note: The practical upper limit depends on the amount of available storage.

Native VSAM: For native VSAM data sets, the buffer page count specifies the number of pages in the buffer used to transfer *data* between memory and auxiliary storage. For LSR buffers, the page count specifies the number of pages used to transfer both *data and index entries*.

OPSYS storage

Places the buffer in a contiguous block of storage acquired from the operating system. The storage is acquired above the 16-megabyte line in operating systems that support extended addressing. If sufficient storage is not available, storage is acquired as IDMS storage. OPSYS STORAGE is the default.

IDMS storage

Acquires a discrete piece of storage for each buffer page. If the operating system supports extended addressing, the storage will be acquired above the 16-megabyte line.

Native VSAM: Do not specify this clause.

CENTRAL VERSION MODE BUFFER

Specifies page counts for the buffer when the database is used under the central version.

cv-buffer-options

Specifies options for the buffer used under the central version.

INITIAL PAGES *initial-page-count*

Specifies the initial number of pages to be allocated for the buffer. *Initial-page-count* is an integer. Valid values appear as follows:

Buffer Type	Valid Values
Non-native VSAM buffers	3 to 16,777,2141; default 3

Buffer Type	Valid Values
Native VSAM buffers	2 to 256; must be greater than the value assigned to STRNO in the NATIVE VSAM clause above

Note: The practical upper limit depends on the amount of available storage.

Native VSAM: For native VSAM data sets, the buffer page count specifies the number of pages in the buffer used to transfer data between memory and auxiliary storage. For LSR buffers, the page count specifies the number of pages used to transfer both data and index entries.

MAXIMUM PAGES *maximum-page-count*

Specifies the largest number of pages that can be allocated for the buffer. *Maximum-page-count* is an integer in the range 3 to 16,777,214. It must be greater than or equal to the number specified in the INITIAL PAGES parameter. The default is the initial number of pages included in the buffer.

Native VSAM: Do not specify this clause.

OPSYS storage

Places the buffer in contiguous storage acquired from the operating system. OPSYS STORAGE is the default.

The storage is acquired above the 16-megabyte line in operating systems which support extended addressing. If sufficient storage is not available, storage is acquired as IDMS storage.

IDMS storage

Requests a discrete piece of storage for each buffer page from the DC/UCF storage pool. If the DC/UCF system contains a storage pool above the 16-megabyte line, then storage for the buffer is acquired above the 16-megabyte line.

Native VSAM: Do not specify this clause.

DETails

Displays or punches details about the database buffer.

HIStory

Displays or punches:

- The user who defined the database buffer
- The user who last updated the database buffer
- The date the database buffer was created
- The date the database buffer was last updated

ALL

Displays or punches all information about the database buffer. ALL is the default action for a DISPLAY or PUNCH verb.

NONe

Displays or punches the name of the database buffer.

Usage

Buffer Storage Not Acquired Until Needed

CA IDMS/DB does not acquire storage for a buffer until it opens a file associated with the buffer.

Buffer Page Count Under the Central Version

When you start up a DC/UCF system, the number of pages in a given buffer is the number specified in the INITIAL PAGES parameter in the buffer definition. If the initial number of pages is lower than the number specified in the MAXIMUM PAGES parameter, you can use the DCMT VARY BUFFER command to increase the number of pages in the buffer up to the specified maximum.

How CA IDMS/DB Acquires Storage for a Buffer

The OPSYS and IDMS parameters tell CA IDMS/DB how to acquire storage for the buffer. In response to the OPSYS parameter, CA IDMS/DB issues a request to the operating system for a contiguous block of storage for the buffer pages. In response to the IDMS parameter, CA IDMS/DB issues requests to the DC/UCF system for storage equal to the size of a buffer page until all the required pages are acquired. For both OPSYS and IDMS, CA IDMS/DB acquires the storage above the 16-megabyte line, if possible.

Dropping a Buffer with Associated Files

Before you delete the definition of a buffer, use the ALTER DMCL statement to change the buffer specification for files associated with the buffer.

Examples

Defining the Default Buffer

The CREATE BUFFER statement below defines a buffer for DMCL IDMSDMCL. The buffer can be used in both local mode and under the central version.

```
create buffer idmsdmcl.index_buffer
   page size 4276
   local mode buffer pages 15
   central version mode buffer
      initial pages 100
      maximum pages 500;
```

Modifying the Page Count for Use Under the Central Version

The following ALTER BUFFER statement modifies both the initial page count and the maximum page count of the INDEX_BUFFER buffer:

```
alter buffer idmsdmcl.index_buffer
   central version mode buffer
      initial pages 150
      maximum pages 300;
```

Dropping a Database Buffer

The following DROP BUFFER statement deletes the definition of the INDEX_BUFFER buffer from the dictionary:

```
drop buffer idmsdmcl.index_buffer;
```

More Information

- For more information about defining database buffers, see Chapter 5, "Defining, Generating, and Punching a DMCL".
- For more information about modifying database buffers, see Chapter 27, “Modifying Physical Database Definitions”.
- For more information about tuning buffers, see Chapter 24, “Monitoring and Tuning Database Performance”.
- For more information about the DCMT VARY BUFFER command, see the *CA IDMS System Tasks and Operator Commands Guide*.

DBGROUP Statements

The DBGROUP statements create, alter, drop, display, or punch a database group definition.

Authorization

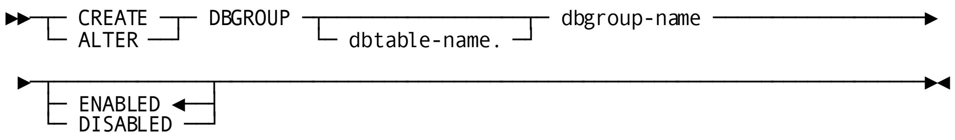
To create, alter, or drop a database group, you must have the following privileges:

- DBADMIN on the dictionary in which the database group definition resides
- ALTER on the database name table in which the database group resides
- CREATE, ALTER, or DROP, respectively, on the database group specified on the DBGROUP statement

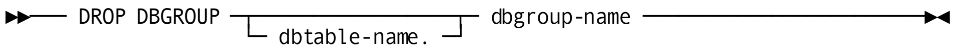
To display or punch a database group, you must hold DISPLAY on the DBGROUP specified in the DBGROUP statement or DBADMIN on the dictionary in which the database name table resides.

Syntax

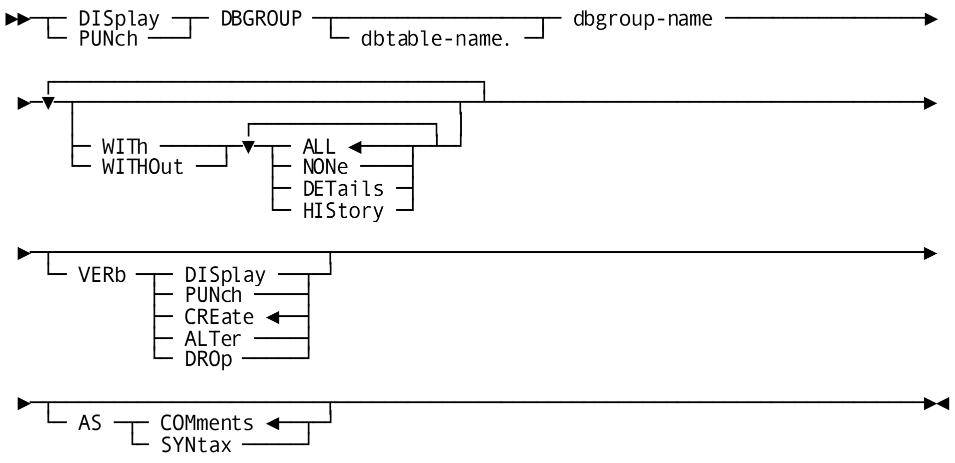
CREATE/ALTER DBGROUP



DROP DBGROUP



DISPLAY/PUNCH DBGROUP



Parameters

dbtable-name

Identifies a database name table defined to the dictionary. You must specify the database name table if you have not established a current database name table as described in 7.3.3, “Entity Currency" earlier in this chapter.

dbgroup-name

Specifies a unique database group in the database name table. *Dbgroup-name* is a 1-8-character value that follows the conventions described in 7.3, “Naming Conventions" earlier in this chapter.

ENABLED/NOT ENABLED

Specifies whether or not an IDMS system using this database name table will become a member of the database group when the system is started. If the system is not a member of the group, it cannot service database requests directed to the specified group.

Once the system is active, group membership status can be changed by issuing a DCMT VARY DBGROUP statement.

DETails

Displays or punches details about the database group.

HIStory

Displays or punches:

- The user who defined the database group
- The user who last updated the database group
- The date the database group was created
- The date the database group was last updated

ALL

Displays or punches all information about the database group. ALL is the default action for a DISPLAY or PUNCH verb.

NONe

Displays or punches the name of the database group.

Usage

What the DBGROUP Statement Does

Each DBGROUP statement defines a database group entry in the database name table. Each DBGROUP statement defines a database group that may be specified in place of a nodename for dynamic routing purposes.

Examples

Defining a Database Group

This example defines two database groups, one representing all CVs that can service customer-related transactions (CUSTGRP) and another that can service finance-related transactions (FINGRP). Both groups have been included in the database name table called CUSTDBT, while only FINGRP has been included in the database name table called CORPDBT.

```
create dbgroup custdbt.custgrp;

create dbgroup custdbt.fingrp;

create dbgroup corpdbt.fingrp;
```

More Information

- For more information about using database name tables when defining a physical database, see Chapter 6, “Defining a Database Name Table”.
- For more information about modifying database name tables, see Chapter 28, “Modifying Database Name Tables".
- For more information about DBGROUPs and dynamic routing, see the *CA IDMS System Operations Guide*.

DBNAME Statements

The DBNAME statements create, alter, drop, display, or punch a database name definition.

Authorization

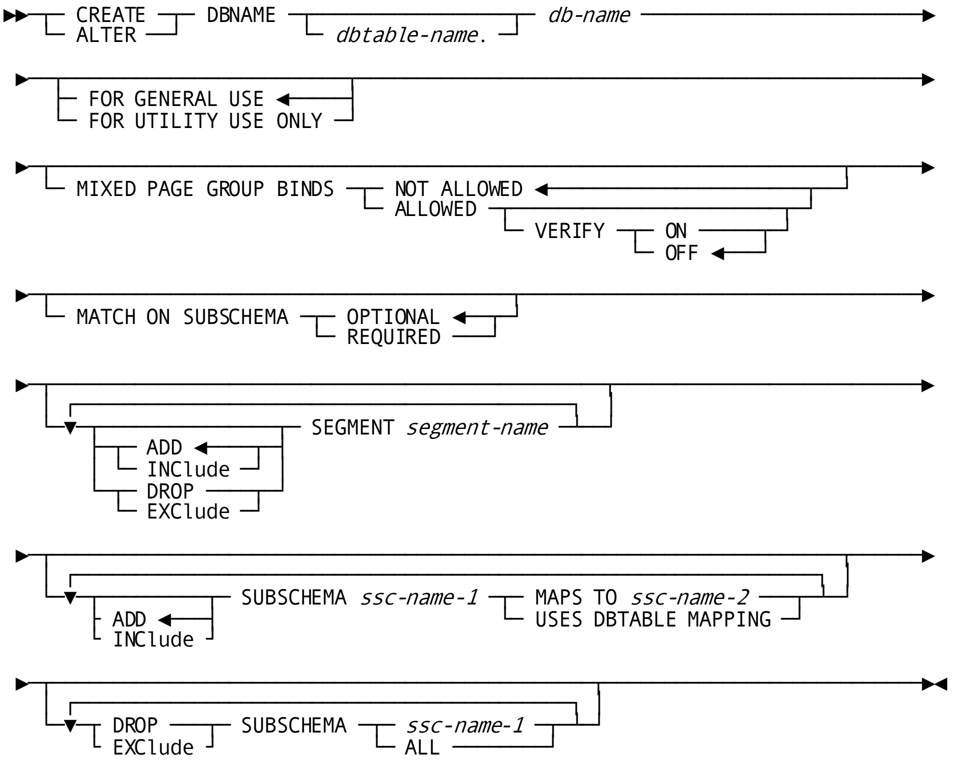
To create, alter, or drop a database name, you must have the following privileges:

- DBADMIN on the dictionary in which the database name definition resides
- ALTER on the database name table in which the database name resides
- CREATE, ALTER, or DROP, respectively, on the database name specified on the DBNAME statement

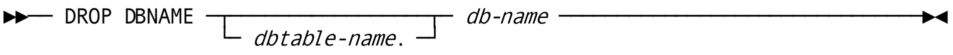
To display or punch a database name, you must hold DISPLAY on the DBNAME specified in the DBNAME statement or DBADMIN on the dictionary in which the database name table resides.

Syntax

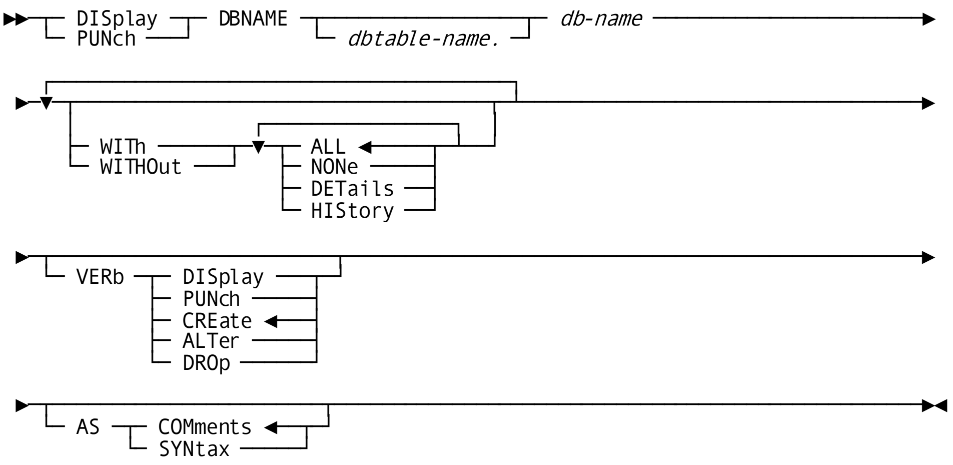
CREATE/ALTER DBNAME



DROP DBNAME



DISPLAY/PUNCH DBNAME



Parameters

dbtable-name

Identifies a database name table defined to the dictionary. You must specify the database name table if you have not established a current database name table as described in 7.3.3, "Entity Currency" earlier in this chapter.

db-name

Specifies a unique database name in the database name table. *Db-name* is a 1- to 8-character value that follows the conventions described in 7.3, "Naming Conventions". It cannot be the reserved keyword '*DEFAULT'.

FOR GENERAL USE

Specifies that the database name is intended for use by application programs.

FOR UTILITY USE ONLY

Specifies that the database name is intended for administrative purposes only.

MIXED PAGE GROUP BINDS ALLOWED|NOT ALLOWED

Specifying MIXED PAGE GROUP BINDS ALLOWED on a DBNAME statement allows a rununit accessing the DBNAME to bind to areas with a mixture of page group and radix values. If not explicitly specified, a rununit binding to a DBNAME whose segments have different page groups will fail if the subschema being used includes areas with different page groups. The default is NOT ALLOWED.

Note: This option applies only to non-SQL-defined databases. Mixedpage group access is always ALLOWED for SQL-defined databases.

VERIFY ON|OFF

Specifies whether or not a check will be made at bind rununit time to ensure that no chain sets included in the subschema cross page group boundaries. If VERIFY OFF is specified, it is your responsibility to ensure that this condition is met. The default for VERIFY is OFF.

Notes:

- This option applies only to non-SQL-defined databases. The VERIFY option is always off for SQL-defined databases.
- A runtime check is always performed for update operations to SQL-defined databases to ensure that the referenced and referencing tables are in the same page group and have the same number of records per page. The VERIFY option setting does not control this runtime check.

MATCH ON SUBSCHEMA OPTIONAL

Specifies that the subschema name passed with the BIND RUN-UNIT statement does not have to be present in the database name definition. OPTIONAL is the default.

MATCH ON SUBSCHEMA REQUIRED

Specifies that the subschema name passed with the BIND RUN-UNIT statement must be present in the database name definition. If the subschema name is not present, the bind is rejected.

ADD SEGMENT

Associates a segment with the database name. ADD is the default. You have to add at least one segment to a database name definition.

DROP SEGMENT

Disassociates a segment from the database name.

segment-name

Identifies a segment to be added to or dropped from the database name definition.

ADD SUBSCHEMA

Adds or updates a subschema mapping associated with the database name. This clause either maps the subschema name passed in a BIND RUN-UNIT statement to the name of a corresponding subschema that CA IDMS/DB will use to access the database or it specifies that the subschema mappings associated with the DBTABLE statement are to be used in determining the database name to be accessed.

Note: New subschema mappings are added at the end of all existing mappings associated with the database name.

ssc-name-1

Specifies the name of a subschema passed in a BIND RUN-UNIT statement. You can use wildcards to specify the subschema name as described below under "Usage".

ssc-name-2

Specifies the name of a subschema to which CA IDMS/DB maps the subschema named in the BIND RUN-UNIT statement. You can use wildcards to specify the subschema name as described below under "Usage".

USES DBTABLE MAPPING

Selects an alternate database name using the subschema name passed on the BIND RUN-UNIT statement and the subschema mapping rules associated with the DBTABLE statement.

DROP SUBSCHEMA

Remove a subschema mapping from the database name definition. *Ssc-name-1* must be the same as that specified in a subschema mapping associated with the database name.

ALL

Removes all subschema mappings from the database name definition. This can be useful when the subschema mappings must be reordered. You can drop all mappings and then re-add them in a different order.

DETails

Displays or punches details about the database name.

HIStory

Displays or punches:

- The user who defined the database name
- The user who last updated the database name
- The date the database name was created
- The date the database name was last updated

ALL

Displays or punches all information about the database name. ALL is the default action for a DISPLAY or PUNCH verb.

NONe

Displays or punches the name of the database name.

Usage

What the DBNAME Statement Does

Each DBNAME statement defines an entry in the database name table. Each DBNAME statement defines a database name that may be specified in a BIND RUN-UNIT or SQL CONNECT statement unless FOR FOR UTILITY USE ONLY is specified. If UTILITY USE ONLY is specified, the database name can only be used for administrative purposes.

Restrictions on Names

The following restrictions apply to database name definitions:

- The database name must be different from any segment included in a DMCL associated with the database name table unless the segment is included in the database name definition.
- The names of all areas associated with segments added to the database name definition must be unique unless FOR UTILITY USE ONLY is specified. For example, you cannot have an area named EMP_AREA in segments EMPSEG and PROJSEG if both segments are included in a database name definition.

These restrictions are checked at runtime. If violated, the database name is marked in error and no transaction will be allowed to access it.

Using Wildcards for Mapping Subschemas

When you specify a subschema name, you can use a question mark (?) to indicate any character. Each question mark in *ssc-name-1* will match any character in the corresponding position of a subschema name passed on the BIND RUN-UNIT statement. For example, an *ssc-name-1* of EMP??T? will match all 7-character subschema names beginning with EMP and having a "T" as the sixth character.

Each question mark in *ssc-name-2* will preserve the character in the corresponding position of the subschema name passed on the BIND. For example, an *ssc-name-2* of EMP??P? will replace the first three characters and the sixth character of the subschema name passed on the bind statement with "EMP" and "P" respectively. The remaining characters of the subschema name remain unchanged. If *ssc-name-2* is ????????, the subschema name passed on the bind statement remains unchanged.

Mapping Sequence Is Important if Using Wildcards

Subschema mappings are searched from top to bottom until a match is found on *ssc-name-1*. Therefore, you should list the most specific subschema mapping first and the least specific last. For example:

```
add subschema emp???? maps to emp????
  .
  .
  .
add subschema ???????? maps uses dbtable mapping
```

Examples

Defining a Database Name

This example defines a production database (EMPDB) and a test database (TESTDB) as entries in database name table ALLDBS. EMPDB contains two segments: EMPSEG containing employee information and PROJSEG containing project information. Similarly, TESTDB contains two segments, TEMPSEG and TPROJSEG containing test employee and project data.

```
create dbname alldbs.empdb
  add segment empseg
  add segment projseg;

create dbname alldbs.testdb
  add segment tempseg
  add segment tprojseg;
```

Using Wildcards to Map Subschemas

In this example, the database name TESTDB is changed to map any subschema name beginning with PROD to a subschema name beginning with TEST. The last 4 characters of the subschema name remain unchanged.

```
alter dbname alldbs.testdb
  add subschema prod???? maps to test????;
```

More Information

- For more information about using database name tables and database names when defining a physical database, see Chapter 6, “Defining a Database Name Table".
- For more information about modifying database name tables, see Chapter 28, “Modifying Database Name Tables".

DBTABLE Statements

The DBTABLE statements perform the following tasks:

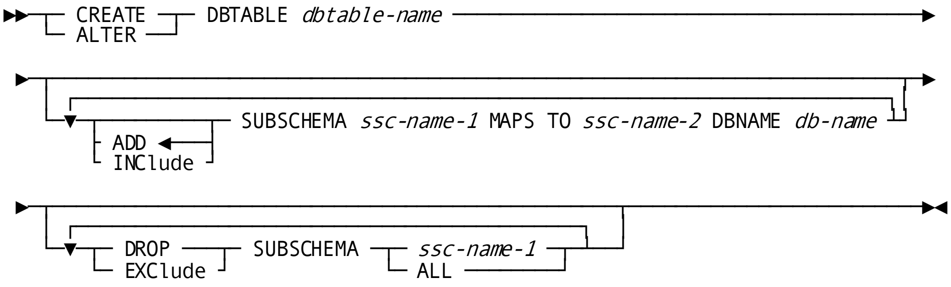
- Creates, alters, drops, displays, or punches a database name table definition in the dictionary
- Generates or deletes a database name table load module in the DDLCATLOD area of the dictionary

Authorization

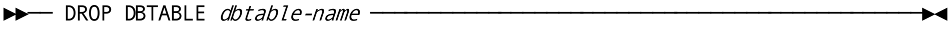
- To create, alter, drop, or generate a database name table, you must have the following privileges:
  - DBADMIN on the dictionary in which the database name definition resides
  - CREATE (for creating), ALTER (for altering or generating), or DROP (for dropping) on the database name table
- To delete the database name table load module, you must have USE authority on the named load module.
- To display or punch the database name table, you must hold DISPLAY privilege on the database name table, or DBADMIN on the dictionary in which the database name table definition resides.

Syntax

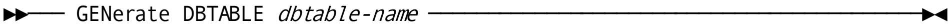
CREATE/ALTER DBTABLE



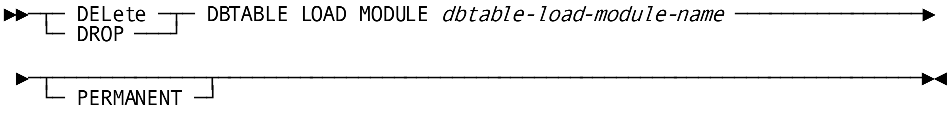
DROP DBTABLE



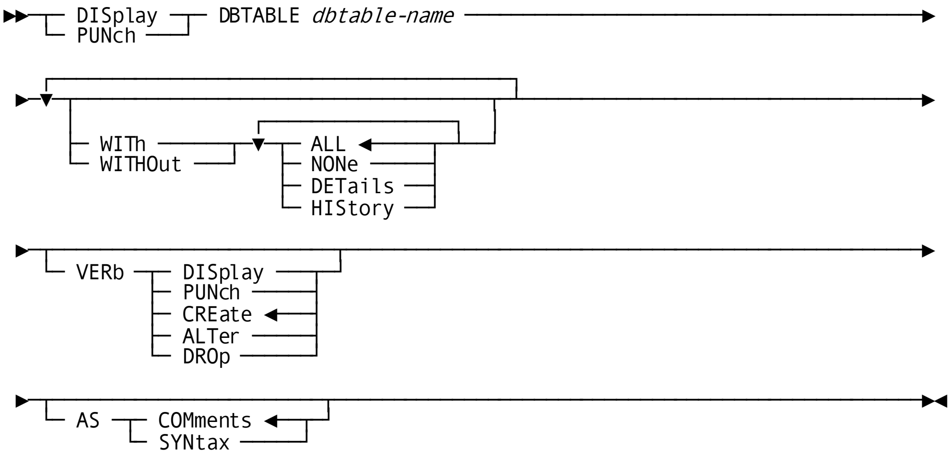
GENERATE DBTABLE



DELETE DBTABLE LOAD MODULE



DISPLAY/PUNCH DBTABLE



Parameters

dbtable-name

Specifies the name of a database name table. *Database-name-table* is a 1- to 8-character value that assigns a unique name to the database name table within the dictionary.

ADD SUBSCHEMA

Identifies the database to be accessed by adding or updating a DBTABLE mapping that maps the name of the subschema specified in a BIND RUN-UNIT statement to a corresponding subschema and its associated database name definition. ADD is the default.

New DBTABLE mappings are added at the end of all existing mappings associated with the database name table.

See "Usage" below for information on using this clause.

ssc-name-1

Specifies a 1- to 8-character name of a subschema passed on a BIND RUN-UNIT statement. You can use wildcards to specify the subschema name as described below under "Usage".

ssc-name-2

Specifies a 1- to 8-character name of a subschema to which CA IDMS/DB maps the subschema named on a BIND RUN-UNIT statement. You can use wildcards to specify the subschema name as described in the "Usage" topic in this section.

db-name

Identifies the database to be accessed. *Db-name* is a 1- to 8-character value that identifies a database name definition in the database name table. See the "Usage" topic in this section for information on how CA IDMS/DB uses this database name at runtime.

DROP SUBSCHEMA

Drops a DBTABLE mapping from the database name table. The name specified in *ssc-name-1* must be the same as that in a subschema mapping associated with the database name table.

ALL

Removes all DBTABLE mappings from the database name table. This can be useful when the mappings must be reordered. You can drop all mappings and then re-add them in a different sequence.

dbtable-load-module-name

Specifies the name of the database name table load module to delete from the DDLCATLOD area.

PERMANENT

Physically erases the database name table load module. By default, IDMS/DB logically erases the database name table load module and physically erases it upon system startup.

DETails

Displays or punches details about the database name table.

HIStory

Displays or punches:

- The user who defined the database name table
- The user who last updated the database name table
- The date the database name table was created
- The date the database name table was last updated

ALL

Displays or punches all information about the database name table. ALL is the default action for a DISPLAY or PUNCH verb.

NONe

Displays or punches the name of the database name table.

Usage

Identify Database Name Table in DMCL

To use the database name table at runtime, you must associate the database name table with the DMCL used at run time.

DBTABLE Mappings Identify Database Names

The primary function of the DBTABLE mappings specified on the DBTABLE statement is to identify the database name to access when none is provided on a BIND RUN-UNIT statement. The subschema mappings are searched for a match on the subschema name passed on the bind. The first subschema mapping with a matching *ssc-name-1* determines the database name to be accessed.

The DBTABLE mappings can also be used if the definition of the database name provided on the bind contains a subschema mapping with the USES DBTABLE MAPPING clause. This clause directs CA IDMS/DB to ignore the database name provided on the bind and to select another database name by using the DBTABLE mappings.

Using Wildcards for Mapping Subschemas

When you specify a subschema name, you can use a question mark (?) to indicate any character. Each question mark in *ssc-name-1* will match any character in the corresponding position of a subschema name passed on the BIND RUN-UNIT statement. For example, an *ssc-name-1* of EMP??T? will match all 7-character subschema names beginning with EMP and having a "T" as the sixth character.

Each question mark in *ssc-name-2* will preserve the character in the corresponding position of the subschema name passed on the BIND. For example, an *ssc-name-2* of EMP??P? will replace the first three characters and the sixth character of the subschema name passed on the bind statement with "EMP" and "P" respectively. The remaining characters of the subschema name remain unchanged. If *ssc-name-2* is ????????, the subschema name passed on the bind statement remains unchanged.

Mapping Sequence Is Important if Using Wildcards

DBTABLE mappings are searched from top to bottom until a match is found on *ssc-name-1*. Therefore, you should list the most specific mapping first and the least specific mapping last. For example:

```
add subschema emp????? maps to emp????? dbname empdb
 .
 .
 .
add subschema ???????? maps to ???????? dbname defdb
```

Generate Creates a Database Name Table Load Module

The GENERATE DBTABLE statement creates and stores a database name table load module. To make a database name table available to CA IDMS/DB you must punch the load module as an object deck and link edit it into the appropriate load library.

To punch a database name table load module as an object deck, use the PUNCH DBTABLE LOAD MODULE utility statement.

Regenerate the Database Name Table Following Changes

You must regenerate the database name table following any additions, changes, or deletions by issuing a GENERATE DBTABLE statement.

Defining the Default Dictionary

One of the primary functions of a database name table is to identify the default dictionary. A default dictionary is the dictionary accessed when you don't specify a dictionary explicitly. It is defined as the database name to which the IDMSNWKL subschema maps. Typically, it is specified using a subschema mapping statement such as:

```
subschema idmsnwk? maps to idmsnwk? dbname defdict
```

You *must* define a default dictionary in every database name table you create.

Examples

Defining a Database Name Table

The following statement creates the ALLDBS database name table. It illustrates the use of DBTABLE mappings to select a database name for processing. All run units binding with a subschema name beginning with CUST will access CUSTDB; those with names beginning with EMP will access the EMPDB; all others will access DEFDB.

```
create dbtable alldbs
   subschema emp????? maps to emp????? dbname empdb
   subschema cust???? maps to cust???? dbname custdb
   subschema ???????? maps to ???????? dbname defdb;
```

Generating a Database Name Table

The following example generates a load module for database name table ALLDBS:

```
generate dbtable alldbs;
```

Identifying the Default Dictionary

This example identifies TESTDICT as the default dictionary. The DBTABLE mapping maps all IDMSNTWK subschemas to dictionary TESTDICT. The dictionary contains segments for the base definition areas, catalog areas and the system message area:

```
create dbtable alldbs
  add subschema idmsnwk? maps to idmsnwk? dbname testdict;
 .
 .
 .
create dbname alldbs.testdict
  add segment testdict
  add segment catseg
  add segment sysmsg;
```

More Information

- For more information about using database name tables, see Chapter 6, “Defining a Database Name Table".
- For more information about modifying database name tables, see Chapter 28, “Modifying Database Name Tables”.
- For more information about establishing a default dictionary, see Chapter 25, “Dictionaries and Runtime Environments".

DISK JOURNAL Statements

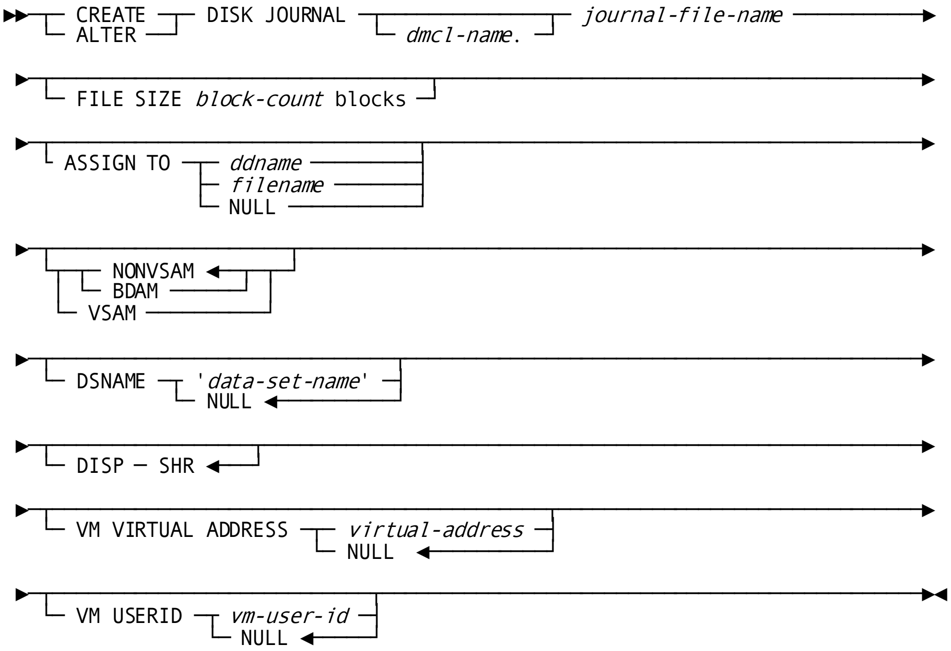
The DISK JOURNAL statements create, alter, drop, display, or punch the definition of a disk journal file from the dictionary.

Authorization

- To create, alter, or drop a disk journal file, you must have the following privileges:
  - DBADMIN on the dictionary in which the disk journal file definition resides
  - ALTER on the DMCL with which the disk journal file is associated
- To display or punch a disk journal file, you must have DISPLAY privilege on the DMCL with which the disk journal file is associated or DBADMIN on the dictionary in which the disk journal file definition resides.

Syntax

CREATE/ALTER DISK JOURNAL



CREATE
ALTER
DISK JOURNAL
dmcl-name.
journal-file-name

FILE SIZE *block-count* blocks

ASSIGN TO
ddname
filename
NULL

NONVSAM
BDAM
VSAM

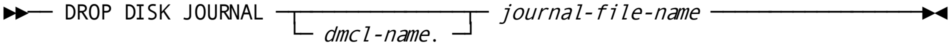
DSNAME
'*data-set-name*'
NULL

DISP – SHR

VM VIRTUAL ADDRESS
virtual-address
NULL

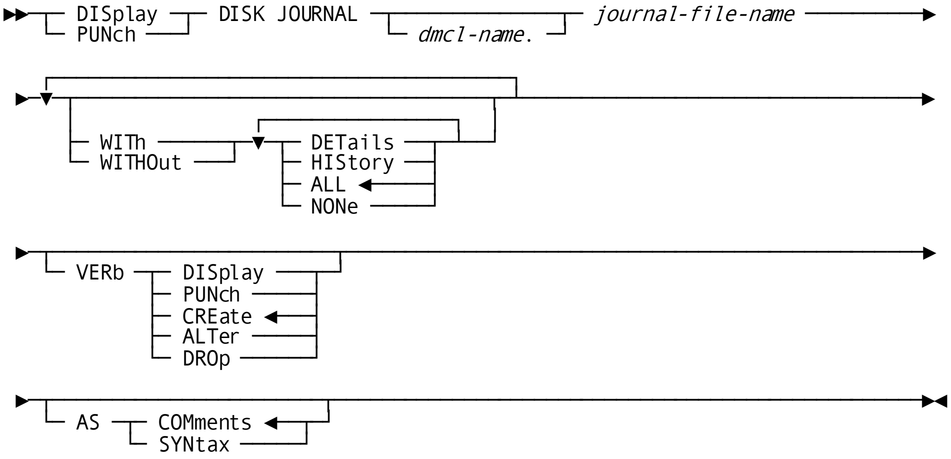
VM USERID
vm-user-id
NULL

DROP DISK JOURNAL



DROP DISK JOURNAL
dmcl-name.
journal-file-name

DISPLAY/PUNCH DISK JOURNAL



DISplay
PUNch
DISK JOURNAL
dmcl-name.
journal-file-name

WITh
WITHOut
DETails
HIStory
ALL
NONe

VERb
DISplay
PUNch
CREate
ALTer
DROp

AS
COMments
SYNtax

Parameters

dmcl-name

Identifies the DMCL with which the disk journal file is associated. *Dmcl-name* must name an existing DMCL defined to the dictionary. If you don't specify a DMCL name, you must establish a current DMCL as described in 7.3.3, "Entity Currency" earlier in this chapter.

journal-file-name

Specifies the name of the journal file. *Journal-file-name* must be a 1- through 18-character name that follows the conventions described in 7.3, "Naming Conventions".

Journal-file-name must be unique among the disk and archive journal file names within the DMCL definition.

FILE SIZE *block-count*

Specifies the number of blocks in the journal file. This clause is required on a CREATE statement. *Block-count* is an integer in the range 9 through 2,147,483,647.

ASSIGN TO

Specifies an external file name. Every external file name in a DMCL definition must be unique. In z/VSE without DYNAM/D, an external file name must be specified. In other environments, if the external file name is not specified, a data set name or VM virtual address must be specified.

ddname

(z/OS and z/VM systems only) Specifies the external name for the file. *ddname* must be a 1- through 8-character value that follows operating system conventions for ddnames.

filename

(z/VSE systems only) Specifies the external name for the file. *filename* must be a 1-through 7-character value that follows operating system conventions for file names.

NULL

Sets the external file name to blanks. It is equivalent to not specifying an external file name for a file. This option is not valid under z/VSE unless DYNAM/D is being used.

NONVSAM

Identifies the access method for the journal file as BDAM, or DAM. BDAM is a synonym for NONVSAM. NONVSAM is the default.

The access method you specify must be the same for all disk journal files associated with the DMCL.

VSAM

Identifies the access method for the journal file as VSAM. The access method you specify must be the same for all disk journal files associated with the DMCL.

DSNAME *data-set-name*

Specifies the name of the data set to be used when dynamically allocating the journal file for z/OS, z/VSE, and OS-format data sets under z/VM.

data-set-name must conform to host operating system rules for forming data set names.

A *data-set-name* that includes embedded periods must be enclosed in single or double quotation marks.

Under z/VM, the DSNAME parameter or VM VIRTUAL ADDRESS and USERID parameters, or both can be specified.

NULL

Sets the data set name to blanks. This is equivalent to not specifying a data set name for a file.

DISP

(z/OS and z/VM systems only) Specifies the disposition to be assigned when the file is dynamically allocated.

SHR

Indicates that the data set used for the file is available to a DC/UCF system and multiple local mode applications at a time.

Under z/VM, DISP SHR causes a link with an access mode of multiple read (MR).

SHR is the default when you do not include the DISP parameter in a CREATE JOURNAL FILE statement.

VM VIRTUAL ADDRESS '*virtual-address*'

(z/VM systems only) Specifies the virtual address of the minidisk used for the journal file. *virtual-address* is a hexadecimal value in the range X'01' to X'FFFF'.

NULL

Sets the virtual address to blanks. On CREATE statements, this is equivalent to not specifying a virtual address for a file. On ALTER statements, it removes any previous virtual address specification for the file.

VM USERID *vm-user-id*

(z/VM systems only) Identifies the owner of the minidisk used for the journal file. *vm-user-id* is a 1- to 8-character value.

A user ID for an OS-format data set must be specified. The user ID is optional for CMS-format files.

If a user ID is not specified for a CMS-format file, CA IDMS assumes that the owner of the minidisk is the user ID of the virtual machine in which it is running.

NULL

On CREATE statements, this is equivalent to not specifying a minidisk owner for a file. On ALTER statements, removes any previous minidisk owner specification for the file.

DETails

Displays or punches details about the disk journal.

HIStory

Displays or punches:

- The user who defined the disk journal
- The user who last updated the disk journal
- The date the disk journal was created
- The date the disk journal was last updated

ALL

Displays or punches all information about the disk journal. ALL is the default action for a DISPLAY or PUNCH verb.

NONe

Displays or punches the name of the disk journal.

Usage

Define Two or More Disk Journal Files

You must define at least two disk journal files when you journal to disk. When one journal file is full, CA IDMS/DB switches to another one. You must use an ARCHIVE JOURNAL utility statement to offload the full journal file.

Dynamic Allocation of Journal Files

Dynamic allocation of files is operating system and file-type dependent. For more information about dynamic file allocation in various operating systems, see 7.14.3, “Usage".

Archive Journal File Requirement

When you journal to disk journal files, you must also define at least one archive journal file to which CA IDMS/DB offloads the contents of a disk journal when it is full.

Incompatibility of Tape and Disk Journal Files

You cannot include the definition of a tape journal file in the DMCL if you include the definitions of disk and archive journal files.

Disk Journaling Used Under the Central Version

To take advantage of the automatic recovery and warmstart capabilities offered under the central version, you must journal to disk.

Disk Journals in Local Mode

A DMCL containing disk journals can be used in local mode but no journaling of database activity is performed. To journal in local mode, use a DMCL that defines a tape journal file instead.

Block size of Disk Journal File

The block size of a disk journal file is determined by the page size of the journal buffer. For VSAM disk journals, the page size of the journal buffer must be the control interval size of the disk journal.

The block size or control interval of the disk journal file must not be larger than the block size of the archive journal file.

Dataspaces Not Supported

The use of dataspaces for journal files is not supported.

Examples

Defining a Disk Journal File

The following CREATE DISK JOURNAL statement defines the disk journal file SYSJRNL1:

```
create disk journal idmsdmcl.sysjrnl1
   file size 1000 blocks
   assign to sysjrnl1;
```

Dropping a Disk Journal File

The following DROP DISK JOURNAL statement deletes the definition of the disk journal file TMPJRNL1 from the dictionary:

```
drop disk journal idmsdmcl.tmpjrnl1;
```

More Information

- On the procedure for defining disk journals, see Chapter 5, "Defining, Generating, and Punching a DMCL".
- On journaling procedures, such as offloading, see Chapter 19, “Journaling Procedures".
- On defining archive journal files, see 7.6, "ARCHIVE JOURNAL Statements".

DMCL Statements

The DMCL statements perform the following tasks:

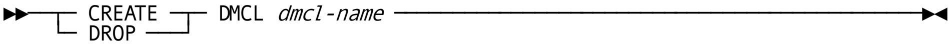
- Creates, alters, or deletes the definition of a DMCL in the dictionary
- Generates a DMCL load module and stores it in the DDLCATLOD area of the dictionary
- Deletes a DMCL load module from the DDLCATLOD area of the dictionary
- Displays or punches the definition of a DMCL in the dictionary

Authorization

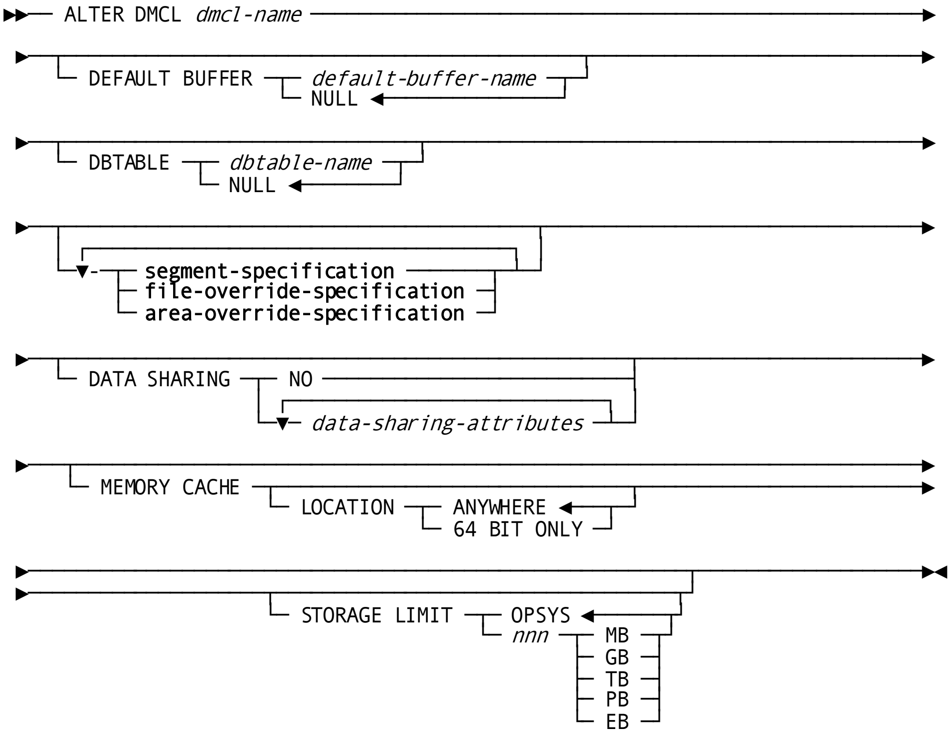
- To create, alter, drop or generate a DMCL, you must have the following privileges:
  - DBADMIN on the dictionary in which the DMCL definition resides
  - CREATE (for creating), ALTER (for altering and generating), or DROP (for dropping) privilege on the named DMCL
  - To alter a DMCL you must have USE authorization on any dbtable including the DMCL
- To delete the DMCL load module, you must have USE authority on the DMCL load module
- To display or punch a DMCL definition, you must have DISPLAY privilege on the named DMCL or DBADMIN authority on the dictionary in which the DMCL definition resides
- To associate a database name table with a DMCL, you must have USE privilege for the named database name table

Syntax

CREATE/DROP DMCL



ALTER DMCL



GENERATE DMCL

```
►►─── GENERATE DMCL dmcl-name ──────────────────────────────────────────►

 ►─┬─────────────────────────┬─────────────────────────────────────────────►◄
   └─ FOR ─┬─ MVS ◄──────────┤
           ├─ VSE ──────────┤
           └─ VM ───────────┘
```

DELETE DMCL LOAD MODULE

```
►►─┬─ DELete ─┬─ DMCL LOAD MODULE dmcl-load-module-name ──────────────►
   └─ DROP ───┘

 ►─┬───────────────┬───────────────────────────────────────────────────►◄
   └─ PERMANENT ───┘
```

DISPLAY/PUNCH DMCL

```
►►─┬─ DISplay ─┬─ DMCL dmcl-name ───────────────────────────────────────►
   └─ PUNch ───┘

    ┌────────────────────────────────────────┐
 ►─▼─┬──────────────────────────────────────┬┴──────────────────────────►
     │                ┌─────────────────┐   │
     └─┬─ WITh ─────┬─▼─┬─ AREas ───────┴─┬─┘
       └─ WITHOut ──┘   ├─ BUFfers ───────┤
                        ├─ FILes ─────────┤
                        ├─ JOUrnals ──────┤
                        ├─ SEGments ──────┤
                        ├─ DETails ───────┤
                        ├─ HIStory ───────┤
                        ├─ ALL ◄──────────┤
                        └─ NONe ──────────┘

 ►─┬──────────────────────────┬─────────────────────────────────────────►
   └─ VERb ─┬─ DISplay ───────┤
            ├─ PUNch ───────┤
            ├─ CREate ◄─────┤
            ├─ ALTer ───────┤
            └─ DROp ────────┘

 ►─┬──────────────────────────┬─────────────────────────────────────────►◄
   └─ AS ─┬─ COMments ◄─────┤
          └─ SYNtax ────────┘
```

Expansion of *data-sharing-attributes*

```
►►─┬─ LOCK ENTRIES lock-entry-count ──────────────┬──────────────────────►◄
   ├─ MEMBERS member-count ───────────────────────┤
   ├─ DEFAULT SHARED CACHE default-cache-name ────┤
   └─ CONNECTIVITY LOSS ─┬─ ABEND ────────────────┘
                         └─ NOABEND ◄─────┘
```

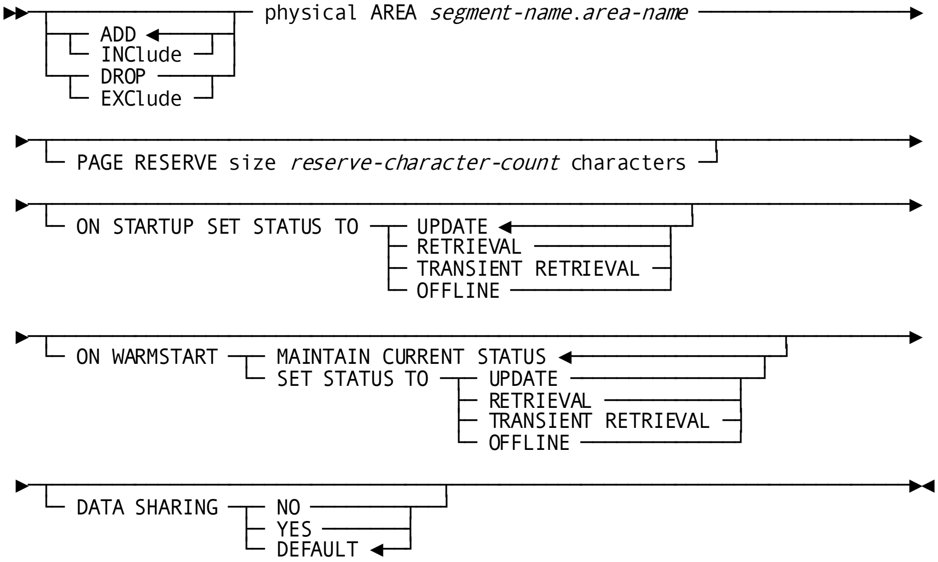
Expansion of *segment-specification*

```
►►─┬───────────────────┬─ SEGMENT segment-name ─────────────────────────►
   ├─┬─ ADD ◄──────┬───┤
   │ └─ INClude ───┘   │
   └─┬─ DROP ──────┬───┘
     └─ EXClude ───┘

 ►─┬────────────────────────────────────────────────┬──────────────────►
   └─ DEFAULT BUFFER ─┬─ database-buffer-name ─┬─────┘
                      └─ NULL ─────────────────┘

 ►─┬──────────────────────────────────────────────────────┬────────────►
   └─ ON STARTUP SET STATUS TO ─┬─ UPDATE ◄──────────────┬─┘
                                ├─ RETRIEVAL ────────────┤
                                ├─ TRANSIENT RETRIEVAL ──┤
                                └─ OFFLINE ──────────────┘

 ►─┬─────────────────────────────────────────────────────────────┬─────►
   └─ ON WARMSTART ─┬─ MAINTAIN CURRENT STATUS ◄──────────────────┬┘
                    └─ SET STATUS TO ─┬─ UPDATE ─────────────────┤
                                      ├─ RETRIEVAL ──────────────┤
                                      ├─ TRANSIENT RETRIEVAL ────┤
                                      └─ OFFLINE ────────────────┘

 ►─┬─────────────────────────────────────────────────────┬─────────────►
   └─ DEFAULT SHARED CACHE ─┬─ default-cache-name ─┬──────┘
                            └─ NULL ◄──────────────┘

 ►─┬─────────────────────────────┬────────────────────────────────────►◄
   └─ DATA SHARING ─┬─ NO ◄──┬───┘
                    └─ YES ──┘
```

Expansion of *file-override-specification*

```
►►─┬───────────────────┬─ FILE segment-name.file-name ──────────────────►
   ├─┬─ ADD ◄──────┬───┤
   │ └─ INClude ───┘   │
   └─┬─ DROP ──────┬───┘
     └─ EXClude ───┘

 ►─┬──────────────────────────────────────────────┬────────────────────►
   └─ BUFFER ─┬─ database-buffer-name ─┬───────────┘
              └─ DEFAULT ◄─────────────┘

 ►─┬──────────────────────────────────┬────────────────────────────────►
   └─ ASSIGN TO ─┬─ ddname ──────┬─────┘
                 ├─ filename ────┤
                 ├─ DEFAULT ◄────┤
                 └─ NULL ────────┘

 ►─┬────────────────────────────────────┬──────────────────────────────►
   └─── DISP ─┬─ SHR ─────────┬──────────┘
              ├─ OLD ─────────┤
              └─ DEFAULT ◄────┘

 ►─┬─────────────────────────────┬─────────────────────────────────────►
   └─ MEMORY CACHE ─┬─ NO ◄──┬───┘
                    └─ YES ──┘

 ►─┬──────────────────────────┬────────────────────────────────────────►
   └─ DATASPACE ─┬─ NO ◄──┬───┘
                 └─ YES ──┘

 ►─┬───────────────────────────────────────┬──────────────────────────►◄
   └─ SHARED CACHE ─┬─ cache-name ──┬───────┘
                    ├─ NULL ────────┤
                    └─ DEFAULT ◄────┘
```

Expansion of *area-override-specification*



```
►►─┬─────────────────────┬─ physical AREA segment-name.area-name ─────────►
   ├─┬ ADD ◄───────────┬─┤
   │ └ INClude ────────┘ │
   └─┬ DROP ───────────┬─┘
     └ EXClude ────────┘

 ►─┬──────────────────────────────────────────────────────────┬──────────►
   └─ PAGE RESERVE size reserve-character-count characters ───┘

 ►─┬──────────────────────────────────────────────────────────┬──────────►
   └─ ON STARTUP SET STATUS TO ─┬─ UPDATE ◄────────────────┬──┘
                                ├─ RETRIEVAL ──────────────┤
                                ├─ TRANSIENT RETRIEVAL ────┤
                                └─ OFFLINE ────────────────┘

 ►─┬──────────────────────────────────────────────────────────┬──────────►
   └─ ON WARMSTART ─┬─ MAINTAIN CURRENT STATUS ◄───────────┬──┘
                    └─ SET STATUS TO ─┬─ UPDATE ──────────┬┘
                                      ├─ RETRIEVAL ───────┤
                                      ├─ TRANSIENT RETRIEVAL ┤
                                      └─ OFFLINE ─────────┘

 ►─┬──────────────────────────────────┬──────────────────────────────────►◄
   └─ DATA SHARING ─┬─ NO ────────────┘
                    ├─ YES ─────────┤
                    └─ DEFAULT ◄────┘
```

Parameters

dmcl-name

Names the DMCL. *Dmcl-name* is a 1- to 8-character name assigned according to naming conventions described in 7.3, "Naming Conventions".

DEFAULT BUFFER *buffer-name*

Specifies the default buffer for the DMCL. *Buffer-name* must identify a database buffer defined in the dictionary and associated with the DMCL.

The default buffer is used for all files, unless overridden at the segment or file level.

Native VSAM: For more information about assigning buffers for native VSAM files, see the "Usage" topic in this section.

NULL

On an ALTER DMCL statement, removes the named buffer as the default buffer for the DMCL.

DBTABLE *dbtable-name*

Specifies the name of the database name table to be used with the DMCL at runtime.

NULL

Disassociates the database name table from the DMCL.

DATA SHARING

Specifies or removes attributes associated with data sharing operations.

- **NO**—Removes data sharing-related information from the DMCL
- *data-sharing-attribute*—Adds or changes the specified data sharing attribute

Data sharing attributes apply to any DC/UCF system that uses this DMCL and is a member of a data sharing group. If data sharing attributes are not included in the DMCL of a CA IDMS system that becomes a member of a data sharing group, the following defaults will be used:

- lock-entry-count: 4096
- member-count: 7
- default-cache-name: null
- connectivity loss: NOABEND

data-sharing-attribute

LOCK ENTRIES *lock-entry-count*

Specifies the number of lock table entries that will be allocated within the coupling facility lock structure. The value specified must be in the range 4096 through 1,073,741,824. The number of lock entries will be rounded up to a power of 2.

MEMBERS *member-count*

Specifies the maximum number of CA IDMS systems that can be members of the system's data sharing group. The value specified must be in the range 7 through 247.

DEFAULT SHARED CACHE *default-cache-name*

Specifies the default shared cache for any system using this DMCL. *Default-cache-name* must identify an XES cache structure defined to a coupling facility accessible to the CA IDMS system.

The default shared cache for a system is used at runtime for any file whose area is designated as shared, if the file does not have an assigned cache. This value has no affect on files that are not associated with a shared area.

ON CONNECTIVITY LOSS

Specifies what action the CA IDMS system is to take when either a loss in connectivity to or a failure of a critical coupling facility structure associated with a data sharing group is detected.

- **ABEND**—Specifies that the CA IDMS system is to abnormally terminate immediately.
- **NOABEND**—Specifies that the CA IDMS is to remain active in order to service non-data sharing-related requests.

NOABEND is the default if ON CONNECTIVITY LOSS is not specified.

MEMORY CACHE

Indicates global options for caching files in memory.

Note: For more information about operating-specific considerations in using memory cache and 64-bit storage, see the *CA IDMS System Operations Guide*.

LOCATION

Indicates where to allocate the storage for memory cache:

ANYWHERE

Memory cache storage is allocated from 64-bit storage; if no or not enough 64-bit storage is available, dataspace storage is acquired.

64 BIT ONLY

Memory cache storage is allocated from 64-bit storage; if no or not enough 64-bit storage is available, memory caching fails.

STORAGE LIMIT

Controls the amount of storage used for memory caching:

OPSYS

Memory cache storage can be acquired until the operating system limit is reached. For 64-bit storage, the operating system limit is set through the MEMLIMIT parameter; for dataspace storage, the limit is optionally imposed by an operating system exit.

***nnn* MB, GB, TB, PB, EB**

CA IDMS controls the amount of memory cache storage if the value specified is smaller than the operating system limit. *nnn* must be a positive value between 1 and 32767. MB, GB, TB, PB, EB indicate the unit in which *nnn* is expressed. The abbreviations stand for Mega Byte (2**20), Giga Byte (2**30), Tera Byte (2**40), Peta Byte (2**50), and Exa Byte (2**60).

segment-specification

On an ALTER DMCL statement, specifies the name of a segment to be added to the DMCL, or identifies a segment in the DMCL to be altered or removed.

ADD

Adds the named segment to the DMCL definition or alters its attributes.

DROP

Drops the named segment from the DMCL definition.

SEGMENT *segment-name*

Identifies the segment. *Segment-name* is a 1- to 8-character value that identifies a segment defined to the dictionary.

DEFAULT BUFFER *buffer-name*

Specifies the buffer to be used by files associated with the segment. *Buffer-name* must identify a buffer associated with the DMCL. Unless overridden by a file override clause, all files associated with the segment will use the named buffer.

Native VSAM: For information about assigning buffers for native VSAM files, see the "Usage" topic in this section.

NULL

Removes the default buffer associated with the segment.

ON STARTUP SET STATUS TO

Specifies the default startup status for areas associated with the segment. The startup status determines how CA IDMS/DB accesses an area when the DC/UCF system is started after an *orderly* shutdown.

The status of an area determines the ready modes in which programs executing under the central version can obtain access to the area.

UPDATE

Sets the status of the area to update and places an external lock on the area.

When the status of an area is update, transactions executing under the central version can obtain access to the area in any ready mode.

ON STARTUP SET STATUS TO UPDATE is the default when you do not include the ON STARTUP parameter in a CREATE SEGMENT statement.

RETRIEVAL

Sets the status of the area to retrieval.

When the status of an area is retrieval, transactions executing under the central version can obtain access to the area in retrieval modes only (that is, transient retrieval, shared retrieval, protected retrieval, and exclusive retrieval).

TRANSIENT RETRIEVAL

Sets the status of the area to transient retrieval.

When the status of an area is transient retrieval, transactions executing under the central version can obtain access to the area only in retrieval ready modes, and regardless of the ready mode, no record or row locks will be acquired.

OFFLINE

Places the area offline.

When the status of an area is offline, transactions executing under the central version cannot obtain access to the area in any ready mode.

ON WARMSTART

Specifies the default warmstart status for areas associated with the segment. The warmstart status determines how CA IDMS/DB accesses an area when the DC/UCF system is started up after an *abnormal termination*.

MAINTAIN CURRENT STATUS

Sets the area status to that in effect at the time the DC/UCF system was abnormally terminated.

ON WARMSTART MAINTAIN CURRENT STATUS is the default when you do not include the ON WARMSTART parameter in a CREATE SEGMENT statement.

DEFAULT SHARED CACHE

Specifies or removes the default shared cache for a segment.

- *default-cache-name*—Specifies the name of the shared cache to be used for files associated with the segment. *Default-cache-name* must identify an XES cache structure defined to a coupling facility accessible to the CA IDMS system.
- **NULL**—Removes the default shared cache from the segment.

NULL is the default if DEFAULT SHARED CACHE is not specified.

The value established at the segment level may be overridden at the file level.

DATA SHARING

Specifies whether or not areas associated with the segment are eligible to be concurrently updated by CA IDMS systems that are members of a data sharing group.

- **YES**—Specifies that concurrent update is allowed.
- **NO**—Specifies that concurrent update is not allowed.

NO is the default if DATA SHARING is not specified.

The value established at the segment level may be overridden for individual areas within the segment.

file-override-specification

On an ALTER DMCL statement, specifies override attributes for a file in a segment that has been added to the DMCL.

ADD

Adds or modifies file override information in the DMCL. ADD is the default.

DROP

Drops file override information from the DMCL.

Note: This parameter does not drop file definitions from the DMCL.

segment-name.file-name

Identifies the file whose attributes are being overridden. *Segment-name* must identify a segment included in the DMCL. *File-name* must identify a file in the named segment.

BUFFER *buffer-name*

Specifies the buffer for the file. *Buffer-name* must identify a buffer associated with the DMCL.

If no buffer is specified on a file override, the default buffer for the segment is used.

Native VSAM: For information about assigning buffers for native VSAM files, see the "Usage" topic in this section.

DEFAULT

Specifies that the file is to use the segment's default buffer. If the segment lacks a default buffer assignment, the default buffer is the default buffer assigned to the DMCL. DEFAULT is the default.

ASSIGN TO

Associates the database file with an external file name that overrides the external file name assigned on a CREATE or ALTER FILE statement. All external file names in a DMCL definition must be unique.

ddname

Specifies the external name for the file under z/OS or z/VM. *Ddname* must be a 1- through 8-character value that follows operating system conventions for ddnames.

filename

Specifies the external name for the file under z/VSE. *Filename* must be a 1- through 7-character value that follows operating system conventions for filenames.

DEFAULT

Removes the external file name override assigned to the file and re-assigns the external file name specified on a CREATE or ALTER FILE statement.

NULL

Removes any external file name for the file. If you specify NULL, you must specify the data set name on the DSNAME clause and/or z/VM VIRTUAL ADDRESS clause of the FILE statement. This option is not valid under z/VSE unless DYNAM/D is being used.

DISP

For z/OS and z/VM systems, specifies the disposition to be assigned when the file is dynamically allocated.

SHR

Indicates that the data set specified on the DSNAME parameter will be available to multiple DC/UCF systems and local mode transactions at a time.

Under z/VM, DISP SHR causes a link with an access mode of multiple read (MR).

OLD

Indicates that the data set specified on the DSNAME parameter will be available to only one DC/UCF system or local mode transaction at a time.

Under z/VM, DISP OLD causes a link with an access mode of multiple write (MW).

MEMORY CACHE NO

Specifies that a file is not to be cached in memory.

MEMORY CACHE YES

Specifies that a file is to be cached in memory.

Note: For more information about operating-system specific considerations in using memory cache and 64-bit storage, see the *CA IDMS System Operations Guide*.

DATASPACE NO

Same as MEMORY CACHE NO. This syntax is provided for upward compatibility only.

DATASPACE YES

Same as MEMORY CACHE YES. This syntax is provided for upward compatibility only.

SHARED CACHE

Specifies or removes the shared cache for a file.

- *cache-name*—Specifies the name of the shared cache to be used for the file. *Cache-name* must identify an XES cache structure defined to a coupling facility accessible to the CA IDMS system.
- **NULL**—Removes the shared cache assigned to the file.
- **DEFAULT**—Specifies that the default cache specified for the segment will be used for the file.

DEFAULT is the default if SHARED CACHE is not specified.

area-override-specification

On an ALTER DMCL statement, specifies override attributes for an area in a segment that has been added to the DMCL.

ADD

Adds or modifies area override information in the DMCL. ADD is the default.

DROP

Drops area override information from the DMCL.

Note: This parameter does not drop area definitions from the DMCL.

segment-name.area-name

Identifies the area whose attributes are being overridden. *Segment-name* must identify a segment included in the DMCL. *Area-name* must identify an area in the named segment.

PAGE RESERVE SIZE *reserve-character-count*

Specifies the number of bytes to be reserved on each page to accommodate increases in the length of record occurrences or rows stored on the page. This clause overrides the value specified in the PAGE RESERVE SIZE clause of a CREATE or ALTER AREA statement.

Reserve-character-count must be either 0 or a multiple of 4 in the range 48 through 32,716 and must be less than or equal to the area's page size. The default is 0.

Native VSAM: For areas defined for native VSAM files, *reserve-character-count* must be 0.

ON STARTUP SET STATUS TO

Specifies a startup status for the area that overrides the startup status specified for the segment with which the area is associated. See above for a description of this clause and its options.

ON WARMSTART

Specifies a warmstart status for the area that overrides the warmstart status specified for the segment with which the area is associated. See above for a description of this clause and its options.

DATA SHARING

Specifies whether or not the area is eligible to be concurrently updated by CA IDMS systems that are members of a data sharing group.

- **YES**—Specifies that concurrent update is allowed.
- **NO**—Specifies that concurrent update is not allowed.
- **DEFAULT**—Specifies that the data sharing attribute of the segment will apply to the area.

DEFAULT is the default if DATA SHARING is not specified.

FOR

Specifies the operating system for which the DMCL is being generated. If not specified, the default is the operating system in which the GENERATE statement is executed.

MVS

Generates a DMCL load module for the z/OS operating system.

VSE

Generates a DMCL load module for the z/VSE operating system.

VM

Generates a DMCL load module for the z/VM operating system.

dmcl-load-module-name

Specifies the name of the DMCL load module to delete from the DDLCATLOD area.

PERMANENT

Physically erases the DMCL load module. By default, CA IDMS/DB logically erases the DMCL load module and physically erases it upon system startup.

AREas

On DISPLAY/PUNCH requests, identifies all database areas defined to the DMCL which have override specifications.

BUFfers

On DISPLAY/PUNCH requests, identifies all database buffers and journal buffers associated with the DMCL.

FILes

On DISPLAY/PUNCH requests, identifies all files defined to the DMCL which have override specifications.

JOUrnals

On DISPLAY/PUNCH requests, identifies all disk, tape, and archive journal files associated with the DMCL.

SEGments

On DISPLAY/PUNCH requests, identifies all segments contained in the DMCL.

DETails

Displays or punches details about the DMCL.

HIStory

Displays or punches:

- The user who defined the DMCL
- The user who last updated the DMCL
- The date the DMCL was created
- The date the DMCL was last updated

ALL

Displays or punches all information about the DMCL. ALL is the default action for a DISPLAY or PUNCH verb.

NONe

Displays or punches the name of the DMCL.

Usage

Ordering Definitions

You must define one or more database buffers for the DMCL before you add segments.

Assigning Buffers for Native VSAM Files

The following restrictions apply to buffers assigned to native VSAM files:

- If the buffer is defined as NSR, all files using it must be associated with a single area.
- If the file access method is KSDS, ESDS, PATH, or RRDS, then the associated buffer must be defined as NSR or LSR. Likewise, if the buffer is defined as NSR or LSR, only KSDS, ESDS, PATH, and RRDS files can use it
- All PATH files associated with an area mapped to a KSDS or ESDS file must use the same buffer as the KSDS or ESDS file.

Assigning Buffers for Other Files

The page size of the buffer must be greater than or equal to the page size of all areas whose files are assigned to the buffer. If the file's access method is VSAM, the page size of the buffer must be greater than or equal to the file's control interval size.

External File Name

All non-blank external file names, including those for both database and journal files, must be unique within a DMCL. If necessary, use file overrides to assign unique names.

An external file name must be specified unless dynamic allocation will be used to access the file.

Note: For more information about dynamic file allocation in various operating systems, see 7.14.3, “Usage".

Archive Journal Block Size

Upon generation, the block size associated with an archive journal is checked to ensure it is not less than the block size of the disk journals. Since the block size of the disk journals is derived from the page size of the journal buffer, if the archive journal's block size is less than the page size of the journal buffer, the page size of the journal buffer is used and a warning message issued.

Caching Files in Memory

You can reduce retrieval I/O operations by caching a file in memory using the MEMORY CACHE clause of the file override specification. File caching is not supported for native VSAM files.

Note: For more information about using memory caching, see Reducing I/O.

Dataspace Versus Memory Cache

The MEMORY CACHE clause replaces the use of the DATASPACE clause. The latter is still accepted for upward compatibility, but is no longer generated on displays.

Controlling Memory Cache

Use the DMCL-wide MEMORY CACHE options to control where and how much memory cache storage can be allocated.

Insufficient Storage for Memory Cache

If MEMORY CACHE YES is specified for a file, and not enough storage is available to cache the file in memory, processing continues as if MEMORY CACHE NO was specified.

Dynamically Changing Memory Cache Specification

The MEMORY CACHE specification can be changed dynamically:

- Use DCMT VARY DMCL to change DMCL-wide MEMORY CACHE options
- Use DCMT VARY FILE to change the MEMORY CACHE specification for a file.

Note: For more information about DCMT VARY DMCL and DCMT VARY FILE, see the *CA IDMS System Tasks and Operator Commands Guide*.

Specifying Data Sharing Attributes

Each data sharing group has an associated coupling facility lock structure. The first CA IDMS system to become a member of the group, establishes the attributes of the lock structure. These attributes remain in effect until all members of the group have terminated normally. As long as any CA IDMS system is either active or has failed and not yet been restarted, the existing lock structure attributes remain in effect. Lock structure attributes include the number of lock entries and the maximum number of members. Both of these attributes affect the size requirements for the lock structure and should be chosen carefully.

Note: For more information about specifying data sharing attributes, see 24.4, "Reducing I/O". Also see the *CA IDMS System Operations Guide*.

Examples

Creating a DMCL

The following statement creates DMCL IDMSDMCL:

```
create dmcl idmsdmcl;
```

Assigning Buffers

The following statement assigns buffers to files associated with the DMCL:

- File INDX_FILE in segment EMPSEG uses INDX_BUFF as its buffer
- All other files in segment EMPSEG use EMP_BUFF as their buffer
- All files in other segments in the DMCL use the default buffer

```
alter dmcl idmsdmcl
     default buffer def_buff
     add segment projseg
     add segment empseg
         default buffer emp_buff
         file empseg.indx_file
         buffer indx_buff
     add segment payseg;
```

More Information

- For more information about the procedure for defining a DMCL, see Chapter 5, “Defining, Generating, and Punching a DMCL”.
- For more information about maintaining a DMCL, see Chapter 27, “Modifying Physical Database Definitions”.
- For more information about specifying data sharing attributes, see Chapter 5, “Defining, Generating, and Punching a DMCL”.
- For more information about memory cache, see 24.4, “Reducing I/O”.
- For more information about data sharing, see the *CA IDMS System Operations Guide*.

FILE Statements

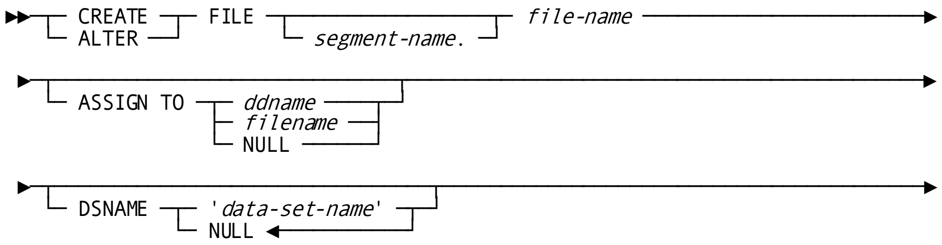
The FILE statements create, alter, drop, display, or punch the definition of a database file in the dictionary.

Authorization

- To create, alter, or drop a database file, you must have the following privileges:
  - DBADMIN on the dictionary in which the file definition resides
  - ALTER privilege on the segment with which the file is associated
- To display or punch a file definition, you must have DISPLAY privilege on the segment with which the file is associated or DBADMIN on the dictionary in which the file definition resides

Syntax

CREATE/ALTER FILE



```
►──┬─────────────────────────────────────────────────────────────────────►
   └─ DISP ─┬─ SHR ◄─────┬┘
            └─ OLD ──────┘

►──┬─────────────────────────────────────────────────────────────────────►
   └─ VM VIRTUAL ADDRESS ─┬─ 'virtual-address' ─┬─┘
                          └─ NULL ──────────────┘

►──┬─────────────────────────────────────────────────────────────────────►
   └─ VM USERID ─┬─ vm-user-id ─┬─┘
                 └─ NULL ───────┘

►──┬──────────────────────────────────────────────┬──────────────────────►◄
   ├─┬─ NONVSAM ◄─┬───────────────────────────────┤
   │ └─ BDAM ─────┘                               │
   ├─ VSAM ───────────────────────────────────────┤
   ├─ ESDS ───────────────────────────────────────┤
   ├─ RRDS ───────────────────────────────────────┤
   └─┬─ KSDS ─┬─┬──────────────┬─┬────────────────────┬─┘
     └─ PATH ─┘ └─ FOR CALC ───┘ └─ FOR SET set-name ─┘
```

DROP FILE

```
►►── DROP FILE ─┬───────────────────┬─ file-name ─────────────────────────►◄
                └─ segment-name. ───┘
```

DISPLAY/PUNCH FILE

```
►►─┬─ DISplay ─┬─ FILE ─┬───────────────────┬─ file-name ─────────────────►
   └─ PUNch ───┘        └─ segment-name. ───┘

   ┌─────────────────────────────────────────┐
►──▼─┬───────────────────────────────────────┴─┬─────────────────────────►
     │                   ┌───────────────┐     │
     └─┬─ WITh ────┬─────▼─┬─ AREas ─────┴─────┘
       └─ WITHOut ─┘       ├─ DETails ───┤
                           ├─ HIStory ───┤
                           ├─ ALL ◄──────┤
                           └─ NONe ──────┘

►──┬─────────────────────────────────────────────────────────────────────►
   └─ VERb ─┬─ DISplay ───┬─┘
            ├─ PUNch ─────┤
            ├─ CREate ◄───┤
            ├─ ALTer ─────┤
            └─ DROp ──────┘

►──┬─────────────────────────────────────────────────────────────────────►◄
   └─ AS ─┬─ COMments ◄─┬─┘
          └─ SYNtax ────┘
```

Parameters

segment-name

Specifies the segment associated with the file. *Segment-name* must identify an existing segment defined to the dictionary.

If you do not specify the segment name, you must establish a current segment as described in 7.3.3, "Entity Currency".

file-name

Specifies the name of the file. *File-name* must be a 1- through 18-character name that follows the conventions described in 7.3, "Naming Conventions".

File-name must be unique within the segment associated with the file.

ASSIGN TO

Specifies an external file name. Every external file name in a DMCL definition must be unique. If you do not specify an ASSIGN TO clause, you must do one of two things:

- Specify the external file name in a file override clause in every DMCL in which the segment is included
- Specify DSNAME or VM VIRTUAL ADDRESS parameter

In z/VSE without DYNAM/D, every file must have an external file name. In other environments, if the external file name is not specified, a data set name or VM virtual address must be specified.

ddname

Specifies the external name for the file under z/OS or z/VM. *Ddname* must be a 1- through 8-character value that follows operating system conventions for ddnames.

filename

Specifies the external name for the file under z/VSE. *Filename* must be a 1- through 7-character value that follows operating system conventions for filenames.

NULL

Sets the external file name to blanks. This is equivalent to not specifying an external file name for a file. This option is not valid under z/VSE unless DYNAM/D is being used.

DSNAME ***data-set-name***

For z/OS and z/VSE and OS-format data sets under z/VM, specifies the name of the data set to be used when dynamically allocating the file. You must include this parameter if the file has no external file name assigned.

Data-set-name must conform to host operating system rules for forming data set names.

A data set name that includes embedded periods must be enclosed in single or double quotation marks.

Under z/VM, you can specify the DSNAME parameter or VM VIRTUAL ADDRESS and USERID parameters, or both.

Note: For more information about allocating files dynamically under z/VSE and z/VM, see the "Usage" topic in this section.

NULL

In ALTER statements, removes any previous data-set name specification for the file.

DISP

For z/OS and z/VM systems, specifies the disposition to be assigned when the file is dynamically allocated.

OLD

Indicates that the data set used for the file will be available to only one DC/UCF system or local mode application at a time.

Under z/VM, DISP OLD causes a link with an access mode of multiple write (MW).

SHR

Indicates that the data set used for the file will be available to multiple DC/UCF systems and local mode applications at the same time.

Under z/VM, DISP SHR causes a link with an access mode of multiple read (MR).

SHR is the default when you do not include the DISP parameter in a CREATE FILE statement.

VM VIRTUAL ADDRESS *'virtual-address'*

For z/VM systems, specifies the virtual address of the minidisk to be used for the file. *Virtual-address* is a hexadecimal value in the range X'0001' to X'FFFF' with all four digits specified.

VM VIRTUAL ADDRESS NULL

On ALTER statements, removes any previous virtual address specification for the file.

VM USERID *vm-user-id*

For z/VM systems only, identifies the owner of the minidisk to be used for the file. *Vm-user-id* is a 1- to 8-character value.

You must specify a user ID for an OS-format data set. The user ID is optional for CMS-format files.

If you do not specify a user ID for a CMS-format file, CA IDMS/DB assumes that the owner of the minidisk is the user ID of the virtual machine in which CA IDMS/DB is running.

NULL

Removes any previous minidisk owner specification for the file.

NONVSAM

Identifies the access method for the file as BDAM, or DAM. BDAM is a synonym for NONVSAM. NONVSAM is the default file access method.

VSAM

Identifies the access method for the file as VSAM.

Specify VSAM for VSAM database files.

ESDS

Identifies the structure of a native VSAM file to be accessed by CA IDMS/DB as ESDS (entry-sequenced data set).

RRDS

Identifies the structure of a native VSAM file to be accessed by CA IDMS/DB as RRDS (relative-record data set).

KSDS

Identifies the structure of a native VSAM file to be accessed by CA IDMS/DB as KSDS (key-sequenced data set).

PATH

Identifies a native VSAM path (alternate index) on ESDS or KSDS native VSAM files.

FOR CALC

Specifies that CALC access to records in the area associated with the file is to be translated into either primary key access (for a KSDS file) or alternate index access (for a PATH file). Only 1 file (KSDS or PATH) associated with an area may contain the FOR CALC clause.

FOR SET *set-name*

Indicates that set access for the named set is to be translated into either primary key access (for KSDS file) or alternate index access (for a path file). *Set-name* is the name of a set defined by a schema SET statement with the VSAM INDEX clause. A given *set-name* can be specified in only one FOR SET clause for files within a segment.

AREas

Displays or punches all areas with which the file is associated.

DETails

Displays or punches details about the file.

HIStory

Displays or punches:

- The user who defined the file
- The user who last updated the file
- The date the file was created
- The date the file was last updated

ALL

Displays or punches all information about the file. ALL is the default action for a DISPLAY or PUNCH verb.

NONe

Displays or punches the name of the file.

Usage

Dynamic File Allocation Under z/VSE without DYNAM/D

Under z/VSE without DYNAM/D, dynamic file allocation is used only when moving a file to another location while CV remains active. It is not used when a file is initially opened. To open files, CA IDMS/DB requires the external filename, DLBL, and EXTENT for every file defined in the DMCL. Specifying a DSNAME as part of the file's definition is optional and does not affect how a file is opened.

To move a file to a new location while CV remains active, follow this procedure:

1. Deallocate the file using the DCMT VARY FILE command DEALLOCATE option
2. Add or replace the DLBL and EXTENT information in the SYSTEM standard label group using z/VSE batch facilities
3. Re-allocate the file using the DCMT VARY FILE command ALLOCATE option
4. Open the file using the DCMT VARY FILE command OPEN option

Important! Be careful when you replace the DLBL and EXTENT information in the SYSTEM standard label group. The DLBL and EXTENT information affects all other jobs in the z/VSE system that try to open or close database files with the same filename.

Dynamic File Allocation under z/VSE with DYNAM/D

If using DYNAM/D in z/VSE, the functionality related to dynamic file allocation is similar to that provided in z/OS. If a DSNAME is specified as part of the file's definition and no matching external file name is defined in a label group, CA IDMS/DB (in conjunction with DYNAM/D) dynamically allocates the file by creating label and extent information during the open process.

Dynamic File Allocation Under z/VM

If a dynamically allocated file under z/VM is:

- *An OS-format data set,* the CREATE FILE statement must include the DSNAME, VIRTUAL ADDRESS, and USERID parameters
- *A CMS-format file*:
  - The file must be a reserved file
  - The CREATE FILE statement must include the VIRTUAL ADDRESS parameter

Dropping a File with Associated Areas

Before you delete the definition of a file, use the ALTER AREA statement to:

- Dissociate the file from any areas with pages that map to the file
- Map the dissociated area pages to one or more other files

Examples

Defining a Preallocated File

The CREATE FILE statement below defines the database file INS_FILE. The file must be defined in the JCL used to execute CA IDMS/DB because no dynamic allocation information was provided.

```
create file demoseg.ins_file
   assign to insfile;
```

Defining File to Be Dynamically Allocated

The CREATE FILE statement below defines a database file to be allocated dynamically under z/OS. Since a ddname was specified, execution JCL can be used to override the dataset name at runtime.

```
create file syspub.public4
   assign to syspub04
   dsname 'corp.syspub.public4';
```

Dropping a Database File

The following DROP FILE statement deletes the definition of the INS_FILE file from the dictionary and from all DMCLs with which it is associated:

```
drop file demoseg.ins_file;
```

More Information

- For more information about the procedure for defining files, see Chapter 5, “Defining, Generating, and Punching a DMCL”.
- For more information about modifying files, see Chapter 27, “Modifying Physical Database Definitions”.
- For more information about file management, such as DASD allocation and formatting, see Chapter 17, “Allocating and Formatting Files”.

JOURNAL BUFFER Statements

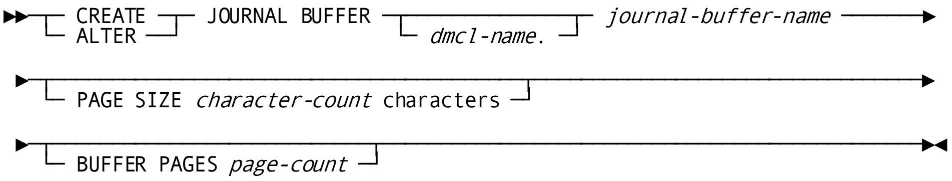
The JOURNAL BUFFER statements create, alter, drop, display, or punch the definition of a journal buffer in the dictionary. For each DMCL, you must define one and only one journal buffer.

Authorization

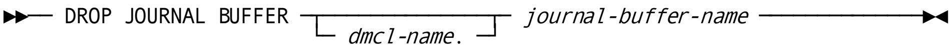
- To create, alter, or drop a journal buffer, you must have the following privileges:
  - DBADMIN on the dictionary in which the journal buffer definition resides
  - ALTER privilege on the DMCL with which the journal buffer is associated
- To display or punch a journal buffer, you must have DISPLAY privilege on the DMCL with which the journal buffer is associated or DBADMIN on the dictionary in which the journal buffer definition resides

Syntax

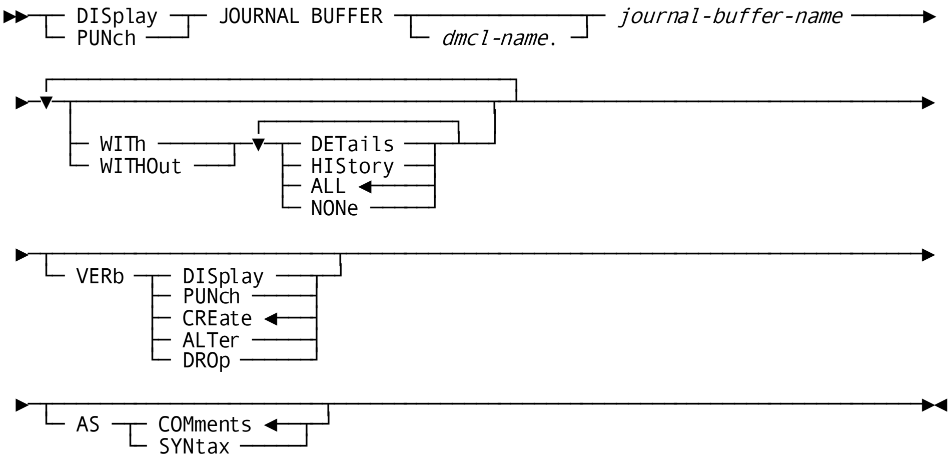
CREATE/ALTER JOURNAL BUFFER



DROP JOURNAL BUFFER



DISPLAY/PUNCH JOURNAL BUFFER



Parameters

dmcl-name

Identifies the DMCL with which the journal buffer is associated. *Dmcl-name* must name an existing DMCL defined to the dictionary. If you don't specify a DMCL name, you must establish a current DMCL as described in 7.3.3, "Entity Currency" earlier in this chapter.

journal-buffer-name

Specifies the name of the journal buffer. *Journal-buffer-name* must be a 1- through 18-character name that follows the conventions described in 7.3, "Naming Conventions".

PAGE SIZE *character-count*

Specifies the number of bytes in each page of the buffer. This clause is required on a CREATE statement. The buffer page size determines the block size for all disk or tape journal files defined in the DMCL. If VSAM disk journals are used, the page size must be 8 bytes less than the file's control interval.

If a page is smaller than 256 bytes, then no data storage is possible. We recommend that a minimum page size of 512 bytes or larger be used.

The value of *character-count* depends upon the operating system:

System	Valid page sizes (in bytes)
z/OS and z/VSE	208 - 32764; multiple of 4. Page size cannot be greater than the maximum block size for the disk device.
z/VM	4096

BUFFER PAGES *page-count*

Specifies the number of pages to be included in the buffer. This clause is required on a CREATE statement. *Page-count* must be an integer in the range 1 through 32,767.

DETails

Displays or punches details about the journal buffer.

HIStory

Displays or punches:

- The user who defined the journal buffer
- The user who last updated the journal buffer
- The date the journal buffer was created
- The date the journal buffer was last updated

ALL

Displays or punches all information about the journal buffer. ALL is the default action for a DISPLAY or PUNCH verb.

NONe

Displays or punches the name of the journal buffer.

Usage

Dropping the Journal Buffer

If you drop the journal buffer associated with a DMCL, be sure to define a new journal buffer before you regenerate the DMCL load module.

Examples

Defining a Journal Buffer

The following CREATE JOURNAL BUFFER statement defines the journal buffer JRNL_BUFF with 3 pages:

```
create journal buffer idmsdmcl.jrnl_buff
   page size 2932 characters
   buffer pages 3;
```

Modifying the Page Size of a Journal Buffer

The following ALTER BUFFER statement changes the page size of journal buffer JRNL_BUFF to 4,352 characters:

```
alter journal buffer idmsdmcl.jrnl_buff
   page size 4352 characters;
```

Dropping a Journal Buffer

The following DROP JOURNAL BUFFER statement deletes the definition of journal buffer JRNL_BUFF from the dictionary:

```
drop journal buffer idmsdmcl.jrnl_buff;
```

More Information

- For more information about the procedure for defining a journal buffer, see Chapter 5, “Defining, Generating, and Punching a DMCL”.
- For more information and considerations about sizing journal buffers, see 5.4.1, “Sizing Journal Buffers”.
- For more information about tuning journal buffer size, see the discussion on Journal use in Chapter 24, “Monitoring and Tuning Database Performance”.
- For more information about journaling procedures, such as offloading, see Chapter 19, “Journaling Procedures”.

SEGMENT Statements

The SEGMENT statements create, alter, drop, display, or punch the definition of a segment in the dictionary.

Authorization

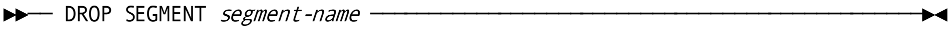
- To create, alter, or drop a segment, you must have the following privileges:
  - DBADMIN on the dictionary in which the segment definition resides
  - CREATE (for creating), ALTER (for altering), or DROP (for dropping) on the named segment
- To display or punch a segment, you must have DISPLAY privilege on the named segment or DBADMIN on the dictionary in which the segment definition resides

Syntax

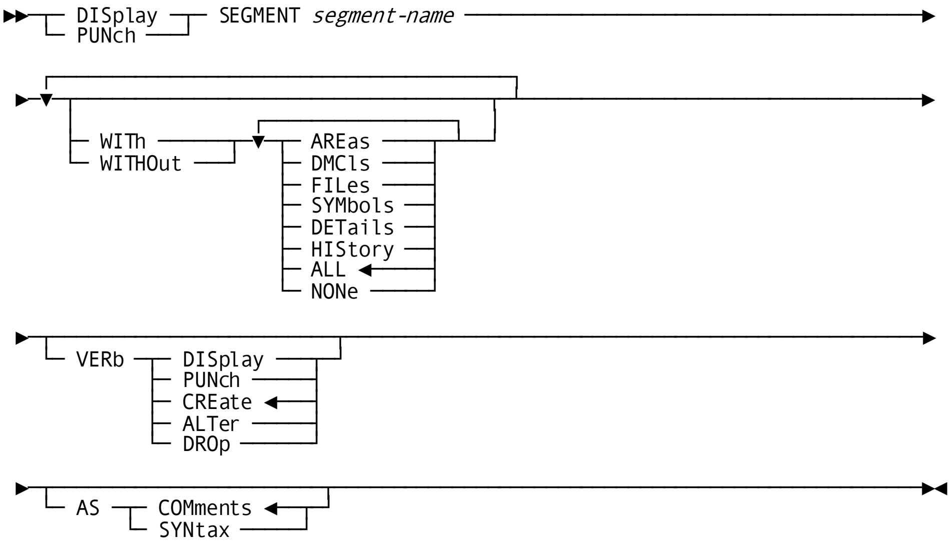
CREATE/ALTER SEGMENT

```
►►─┬─ CREATE ─┬─ SEGMENT segment-name ─────────────────────────────────────►
   └─ ALTER ──┘

►─┬──────────────────────────┬─────────────────────────────────────────────►
  └─ FOR ─┬─ NONSQL ◄──┬─────┘
          └─ SQL ──────┘

►─┬───────────────────────────────────────────────┬────────────────────────►
  └─ PAGE GROUP ─┬─ page-group-number ─┬──────────┘
                 └─ 0 ◄────────────────┘

►─┬──────────────────────────────────────────────────────────────────┬─────►
  └─ MAXIMUM RECORDS PER PAGE ─┬─ maximum-record-count ─┬────────────┘
                               └─ 255 ◄─────────────────┘

►─┬─────────────────────────────────────────────┬──────────────────────────►
  └─ FOR SCHEMA ─┬─ sql-schema-name ─┬──────────┘
                 └─ NULL ────────────┘

►─┬─────────────────────────────┬─────────────────────────────────────────►◄
  └─ STAMP BY ─┬─ TABLE ◄─┬─────┘
               └─ AREA ───┘
```

DROP SEGMENT



DISPLAY/PUNCH SEGMENT



Parameters

segment-name

Specifies the name of the segment. *Segment-name* must be a 1- through 8-character name that follows the conventions described in 7.3, "Naming Conventions".

Segment-name must be unique within the dictionary.

Important! If the segment is an SQL segment in an application dictionary, you must dissociate any tables, indexes, and referential constraints associated with the segment's areas before you attempt to delete the segment by issuing a DROP SEGMENT statement.

FOR NONSQL

Indicates that the segment contains data defined by a non-SQL schema. FOR NONSQL is the default. Valid on CREATE operation only.

FOR SQL

Indicates that the segment contains data defined by an SQL schema. Valid on CREATE operation only.

PAGE GROUP *page-group-number*

Specifies the page group of the segment's areas. *Page-group-number* is an integer in the range 0 through 32767. The default is 0.

MAXIMUM RECORDS PER PAGE *maximum-record-count*

On a CREATE statement, specifies the maximum number of record occurrences that can be stored on a single page of the segment's areas. *Maximum-record-count* is an integer in the range 3 through 2727. The default is 255.

FOR SCHEMA *sql-schema-name*

Reserves areas associated with the segment for tables and indexes in the named SQL schema. *Sql-schema-name* must identify an SQL schema defined in the dictionary or a warning will be issued.

If the segment already contains tables and indexes from other SQL schemas, CA IDMS/DB does not prevent access to them, however, no new ones can be defined.

FOR SCHEMA NULL

On an ALTER statement, removes any previous SQL schema restriction for the segment.

STAMP BY TABLE

For SQL segments only, maintains synchronization stamps at the table level. BY TABLE is the default.

When maintaining stamps at the table level, CA IDMS/DB updates the stamp for an individual table when the definition of the table or any associated calc, index, or constraint is modified.

This clause is ignored for segments defined as non-SQL.

STAMP BY AREA

For SQL segments only, maintains a synchronization stamp at the area level in addition to the stamps maintained for individual tables. When maintaining stamps at the area level, CA IDMS/DB updates the stamps for both the individual table and its area when the definition of any table in the area (or any associated calc, index, or constraint) is modified.

Maintaining stamps at the area level allows validation of access modules by area rather than by individual table.

This clause is ignored for segments defined as non-SQL.

AREas

Displays or punches information about all areas contained in the segment.

DMCLS

Displays or punches information about all DMCLS in which the segment is included.

FILes

Displays or punches information about all files contained in the segment.

SYMbols

Displays or punches information about all symbols defined to areas contained in the segment.

DETails

Displays or punches details about the segment.

HIStory

Displays or punches:

- The user who defined the segment
- The user who last updated the segment
- The date the segment was created
- The date the segment was last updated

ALL

Displays or punches all information about the segment. ALL is the default action for a DISPLAY or PUNCH verb.

NONe

Displays or punches the name of the segment.

Usage

Assigning Page Groups

When you assign a segment to a page group, keep these restrictions in mind:

- For non-SQL defined databases, all data accessed within a run unit must be in the same page group and have the same maximum number of records per page unless you specify the MIXED PAGE GROUP BINDS ALLOWED option
- When adding segments to a DMCL, areas within a page group must have unique, non-overlapping page ranges

CA IDMS/DB Rounds Up the Maximum Record Count

CA IDMS/DB may change the maximum number of records or rows that can be stored on a single page. CA IDMS/DB rounds the value to the next higher power of 2 less 1 to arrive at the actual number of records per page. (This is the largest number that can be represented in the same number of bits.) The following table shows the actual maximum records per page resulting from values specified for *maximum-record-count*.

Value specified in MAXIMUM RECORDS clause	Actual maximum records per page	High allowable page number
3	3	1,073,741,822
4 - 7	7	536,870,910
8 - 15	15	268,435,454
16 - 31	31	134,217,726
32 - 63	63	67,108,862
64 - 127	127	33,554,430
128 - 255	255	16,777,214
256 - 511	511	8,388,606
512 - 1,023	1,023	4,194,302
1,024 - 2,047	2,047	2,097,150
2,048 - 2,727	2,7271	1,048,574

Note: Although a 12-bit line number would theoretically accommodate 4,095 records per page, only 2,727 4-byte record occurrences can actually be stored on the largest possible page.

MAXIMUM RECORDS Clause Determines the Db-Key Format

Because the MAXIMUM RECORDS PER PAGE clause determines the number of bits required for a line number, it also determines the *format of database keys* for the segment. A database key is a 32-bit field, made up of 2 values:

- The number of the page on which a record occurrence or row resides
- The record's or row's line number within that page

Maximum-record-count determines the number of bits required to store a line number (minimum 2 bits; maximum 12); the remaining bits become the page-number portion of the database key. Consequently, *maximum-record-count* and the page numbers assigned to schema areas are dependent upon one another, as is shown in the table above.

In most cases, *maximum-record-count* can be left to default to 255; this accommodates a database with page numbers up to 16,777,214.

Note: The number specified in the MAXIMUM RECORDS clause indicates the *maximum* number of records that the run-time system will place on a single page. The actual number of records on a given page depends on the page size specified on the AREA statement and the sizes of individual records or rows placed on the page.

Note: For information about how the MAXIMUM RECORDS clause and the area's page size affect the number of records or rows on a page, see the presentation of space management in Space Management.

Examples

Defining a Segment

The following CREATE SEGMENT statement defines the SALESSEG segment:

```
create segment salesseg
   for sql
   for schema saleschm
   stamp by area;
```

Dropping a Segment

The following DROP SEGMENT statement deletes the definition of the segment SALESSEG from the dictionary:

```
drop segment salesseg;
```

More Information

- For more information about defining segments, see Chapter 4, “Defining Segments, Files, and Areas”.
- For more information about modifying segments, see Chapter 27, “Modifying Physical Database Definitions”.
- For more information about defining SQL schemas, see Chapter 8, “Defining a Database Using SQL”.

TAPE JOURNAL Statements

The TAPE JOURNAL statements create, alter, drop, display, or punch the definition of a tape journal file in the dictionary. You can define only one tape journal file for any given DMCL.

Authorization

- To create, alter, or drop a tape journal file, you must have the following privileges:
  - DBADMIN on the dictionary in which the tape journal definition resides
  - ALTER privilege on the DMCL with which the tape journal file is associated
- To display or punch a tape journal file definition, you must have DISPLAY privilege on the DMCL with which the tape journal file is associated or DBADMIN on the dictionary in which the tape journal definition resides

Syntax

CREATE/ALTER TAPE JOURNAL

```
►►─┬─ CREATE ─┬─ TAPE JOURNAL ─┬───────────────┬─ journal-file-name ─────────►
   └─ ALTER ──┘                └─ dmcl-name. ──┘

 ►─┬────────────────────────────────┬──────────────────────────────────────►◄
   └─ ASSIGN TO ─┬─ ddname ───────┬─┘
                 └─ filename ─────┘
```

DROP TAPE JOURNAL

```
►►── DROP TAPE JOURNAL ─┬───────────────┬─ journal-file-name ───────────────►◄
                        └─ dmcl-name. ──┘
```

DISPLAY/PUNCH TAPE JOURNAL

```
►►─┬─ DISplay ─┬─ TAPE JOURNAL ─┬───────────────┬─ journal-file-name ───────►
   └─ PUNch ───┘                └─ dmcl-name. ──┘

 ►─▼─┬─────────────────────────────────────────────┬─────────────────────────►
   ┌─┴─────────────────────────────────────────────┘
   │  ├─ WITh ────────┬─▼─┬─ DETails ──┬──┘
   │  └─ WITHOut ─────┘   ├─ HIStory ──┤
   │                      ├─ ALL ◄─────┤
   │                      └─ NONe ─────┘
```

```
►─┬──────────────────────────┬──────────────────────────────────────────►
  └─ VERb ─┬─ DISplay ─────┘
           ├─ PUNch ──────┤
           ├─ CREate ◄────┤
           ├─ ALTer ──────┤
           └─ DROp ───────┘

►─┬───────────────────────┬─────────────────────────────────────────────►◄
  └─ AS ─┬─ COMments ◄──┘
         └─ SYNtax ─────┘
```

Parameters

dmcl-name

Identifies the DMCL with which the tape journal file is associated. *Dmcl-name* must name an existing DMCL defined to the dictionary. If you don't specify a DMCL name, you must establish a current DMCL as described in 7.3.3, "Entity Currency".

journal-file-name

Specifies the name of the tape journal file. *Journal-file-name* must be a 1- through 18-character name that follows the conventions described in 7.3, "Naming Conventions".

ASSIGN TO

Associates the tape journal file with an external file name. This clause is required on a CREATE statement. Each external file name defined to a DMCL must be unique.

ddname

Specifies the external name for the file under z/OS or z/VM. *Ddname* must be a 1- through 8-character value that follows operating system conventions for ddnames.

filename

Specifies the external name for the file under z/VSE. *Filename* must have the following format: SYS*nnn* where *nnn* is a 3-digit number.

DETails

Displays or punches details about the tape journal.

HIStory

Displays or punches:

- The user who defined the tape journal
- The user who last updated the tape journal
- The date the tape journal was created
- The date the tape journal was last updated

ALL

Displays or punches all information about the tape journal. ALL is the default action for a DISPLAY or PUNCH verb.

NONe

Displays or punches the name of the tape journal.

Usage

Mutually Exclusive Journal Definitions

A DMCL must contain the definitions of either disk and archive journal files or a tape journal file. You cannot include the definition of disk and archive journal files in the DMCL if you include the definition of a tape journal file.

Journal File Block Size

The block size of a tape journal file is determined by the page size of the journal buffer associated with the DMCL.

Journaling in Local Mode

If you want to use journaling facilities for a local mode application, the application must use a DMCL in which a tape journal is defined.

Examples

Defining a Tape Journal File

The following CREATE TAPE JOURNAL statement defines the tape journal file TAPEJRNL:

```
create tape journal locdmcl.tapejrnl
   assign to sysjrnl;
```

Changing the External File Name

The following ALTER TAPE JOURNAL statement changes the external file name assigned to tape journal file, TAPEJRNL:

```
alter tape journal locdmcl.tapejrnl
   assign to sysjrnl1;
```

Dropping a Tape Journal File

The following DROP TAPE JOURNAL statement deletes the definition of the tape journal file TAPEJRNL from the dictionary:

```
drop tape journal locdmcl.tapejrnl;
```

More Information

- For more information about defining tape journals, see Chapter 5, “Defining, Generating, and Punching a DMCL”.
- For more information about journaling procedures, see Chapter 19, “Journaling Procedures”.
- For more information about using tape journals for recovery, see 21.2, “Backup Procedures”.

Summary of Physical Database Limits

Data Limits

The following table summarizes the maximum values allowed for physical database definitions:

Item	Maximum allowed
Pages in a data buffer	16,777,214
Bytes in a database buffer page	32,764; multiple of 4

Item	Maximum allowed
Journal buffer pages associated with a database	32,767
Bytes in a journal buffer page	32,768; multiple of 4
Files in a database	32,767
Files associated with an area	32,767
Areas associated with a file	32,767
Pages associated with an area	1,073,741,822
Bytes in a database page	32,764; multiple of 4
Blocks in a disk journal file	999,999
Bytes in an archive journal block	32,768

Chapter 8: Defining a Database Using SQL

This section contains the following topics:

Overview

This chapter contains procedures for defining the logical components of an SQL-defined database (the last step in the list).

Steps to Define a Database

To use SQL to define your database, follow these steps:

1. Design and size the database using information provided in the *CA IDMS Database Design Guide* document.
2. Define in the system dictionary the segments that represent the physical database. Include the segments in your DMCL, and generate, punch, and link edit the DMCL.

   Note: For more information about the physical database, see Chapter 4, “Defining Segments, Files, and Areas".
3. Create and format the operating system files that will contain the table's rows. These files must be accessible to the runtime environment before you define your tables.
4. Copy the segment definition from the system dictionary into the application dictionary in which you wish to define your tables.

   The segment and area names you use in the logical definition must match those defined in the physical definition in the system dictionary. The stamp level, which tells CA IDMS/DB to check the date and time of definition at either the area level or table level, must also match in both definitions. It is recommended that the page range and page size of areas match in both definitions since this information is used for optimization and index sizing. It is not necessary to define the files in the application dictionary.
5. Enter SQL data description (DDL) statements to do the following, in this order:
   - Create the schema
   - Create tables
   - Create CALC keys
   - Create indexes
   - Create referential constraints
   - Drop unneeded default indexes
   - Create views

Note: For complete SQL DDL syntax, see the *CA IDMS SQL Reference Guide*. For design decisions, see the *CA IDMS Database Design Guide* document.

Executing SQL Data Description Statements

Tool for Entering SQL DDL Statements

You enter SQL data description language statements using the online or batch command facility. The command facility performs the following functions:

- Accepts as input SQL data description language (DDL) statements
- Updates the application dictionary with definitions
- Updates the database to reflect the definitions

Note: See the *CA IDMS SQL Programming Guide* for syntax. See the *CA IDMS Common Facilities Guide* document for information about submitting SQL statements using the command facility.

Identifying the Dictionary

When you use the command facility, you must identify the *application dictionary* to be updated by either:

- Explicitly connecting to a dictionary
- Establishing a default dictionary

Executing DDL Statements Programmatically

You can embed SQL DDL statements in an application program. No cursors can be open when you execute embedded DDL statements.

Note: See the *CA IDMS SQL Programming Guide* document for information about embedding SQL statements in an application program.

Local Mode

It is recommended that SQL statements not be executed in local mode. If a local mode error is encountered in the execution of a DDL statement, the dictionary is left in an unpredictable state and *must* be manually recovered. To avoid this, execute SQL DDL statements only under the central version.

Creating a Schema

You create a schema by issuing a CREATE SCHEMA statement.

Things You Can Specify

1. Schema name
2. Optionally a default area
3. Optionally a reference to another schema, either an SQL or non-SQL schema

Considerations

- The default area specified in the CREATE SCHEMA statement must be defined to the application dictionary in which the schema is being defined. The default area is used to contain table rows if no area is specified as part of the table definition.
- If reference is made to another schema, the schema containing the reference is called a referencing schema and the schema that it refers to is a referenced schema. A referencing schema cannot contain table or view definitions.
  - You can reference a non-SQL schema to enable SQL access to a non-SQL defined database.
  - You can reference an SQL schema to allow identical SQL defined databases to be accessed through a single schema definition. The referenced schema must not be itself a referencing schema nor contain tables that reference or are referenced by tables in other schemas. For other considerations associated with referencing SQL schemas, see the *CA IDMS SQL Reference Guide*.
- A referencing schema can be bound to a specific database instance or unbound by not specifying a DBNAME as part of the referencing schema definition. Accessing tables through an unbound referencing schema allows runtime determination of the database instance to be accessed based on the database to which an SQL session connects. Therefore, the same table name (and access modules) can be used to access different database instances by connecting to different database names. Each database name definition must include the appropriate database segments to be accessed.
- The owner of the schema being created (and, therefore, all tables and views within the schema) is the user issuing the CREATE SCHEMA statement. To reassign ownership to another authorization ID, use the TRANSFER OWNERSHIP statement, as described in the *CA IDMS SQL Reference Guide*.

Examples

In the following example, the schema PROD is defined. The default area for the schema is PROD_AREA. Rows in tables associated with this schema will be stored in PROD_AREA unless an area name is explicitly coded in the CREATE TABLE statement.

```
create schema prod
   default area prod_area;
```

In the following example, the schema WINDOW is defined and associated with the non-SQL defined schema SCHED. Programs using SQL data manipulation language statements can access data in the non-SQL database by using the schema WINDOW.

```
create schema window
    for nonsql schema sched;
```

In the following example, the schemas HRTEST1 and HRTEST2 are defined as referencing schemas for SQL schema HRTEST0. References to tables in HRTEST1 will access data in the HRTEST1 database while those in HRTEST2 will access data in the HRTEST2 database. These databases contain identically-defined base tables as described by the HRTEST0 schema.

```
create schema hrtest1
   for sql schema hrtest0 dbname hrtest1;

create schema hrtest2
   for sql schema hrtest0 dbname hrtest2;
```

In the following example, the schema HRTEST is also defined as a referencing schema for SQL schema HRTEST0; however, HRTEST is not associated with any specific database instance. Consequently, the data that is accessed through references to HRTEST tables will be determined at runtime by the database to which the SQL session connects.

```
create schema hrtest
   for sql schema hrtest0;
```

Creating a Table

You create a table by issuing the CREATE TABLE statement and adding appropriate clauses to describe each column associated with the table.

Things You Can Specify

1. Table name, using a schema qualifier unless you have specified a default schema name in the SET SESSION statement

   Note: For more information about session management statements, see the *CA IDMS SQL Reference Guide*.

2. Column names
3. Data type for each column
4. Optionally a default value and a null specification for each column
5. Optionally a check constraint to limit the values allowed in a column or columns
6. An area in which the table's rows will be stored (unless you want them stored in the default area for the schema)
7. Data compression
8. An estimate of the number of rows for the table
9. Physical attributes, including a table ID number and a synchronization timestamp.

Specifying Physical Attributes

When defining or altering a table, you can specify physical attributes that are normally generated automatically. Specifying explicit values for this information, allows you to create tables that have identical physical attributes and can therefore be accessed through a single schema definition.

Since a table's synchronization stamp is updated each time an associated index, calc key or referential constraint is added or removed, the synchronization stamp must be set after adding or removing these associated entities.

Care should be exercised when specifying a specific timestamp, since its purpose is to enable the detection of discrepancies between a table and its definition. If explicitly specified, the timestamp should always be set to a new value following a definitional change so that the change is detectable to the run time system.

Compressing

The COMPRESS option in the table definition statement specifies that data be compressed when it is stored in the database and decompressed when it is retrieved from the database.

To use the COMPRESS option, you must have CA IDMS Presspack installed at your site.

Note: See the *CA IDMS Presspack User Guide* for information about CA IDMS Presspack.

Estimated rows

When you create a new table, it is useful to specify the number of rows you expect to be stored in the table. CA IDMS/DB uses this information to:

- Optimize host language statements that reference the table and are compiled before the table is loaded and the UPDATE STATISTICS statement has been executed for it
- Calculate index characteristics

Example

In the following example, the EMPLOYEE table is defined and associated with the PROD schema. The table includes 15 columns. The check parameter restricts the values that can be inserted in the EMP_ID and STATUS columns. The data in this table will be stored in the EMP.EMPREG area and the expected number of rows for the table is 500.

```
create table prod.employee
                  (emp_id           unsigned numeric      not null,
                   manager_id       unsigned numeric              ,
                   emp_fname        char(20)              not null,
                   emp_lname        char(20)              not null,
                   dept_id          unsigned numeric      not null,
                   street           char(40)                      ,
                   city             char(20)              not null,
                   state            char(02)              not null,
                   zip_code         char(09)              not null,
                   phone            char(10)                      ,
                   status           char                  not null,
                   ss_number        unsigned decimal(9,0) not null,
                   start_date       date                  not null,
                   termination_date date                          ,
                   birth_date       date                          ,
         check ( (emp_id between 0 and 8999) and
                 (status in ('A', 'S', 'L', 'T') ) )
         in emp.empreg
         estimated rows 500;
```

Defining a CALC Key

You create a CALC key by issuing the CREATE CALC statement and specifying a CALC key column.

Things You Can Specify

1. Whether the CALC key is unique
2. Name of the table associated with this CALC key
3. Name of the column or columns that make up the CALC key

Considerations

- You can define only one CALC key for a table.
- The table must be empty when you define a CALC key for it.
- You must specify NOT NULL for the column(s) on which the CALC key is placed if you use the UNIQUE option.
- The table cannot be the referencing table in a clustered referential constraint.
- The table cannot have a clustered index defined on it.

Examples

In the following example, a unique CALC key is defined on the EMPLOYEE table. The CALC key consists of one column, EMP_ID.

```
create unique calc key on prod.employee(emp_id);
```

In the following example, a unique multi-column CALC key is defined on the BENEFITS table. The CALC key consists of two columns, EMP_ID and FISCAL_YEAR.

```
create unique calc key on test.benefits(emp_id, fiscal_year);
```

Defining an Index

You define an index by issuing the CREATE INDEX statement.

Things You Can Specify

1. Whether the index is unique
2. Name of the index
3. Name of the table on which the index is defined
4. Name of the column or columns that make up the index key
5. The sequencing options for the index
6. Optionally, the area in which the index will be stored
7. Optionally, physical characteristics of the index
8. Optionally, physical attributes, including an index ID

Considerations

- CA IDMS/DB will automatically determine the physical characteristics of the index based on the estimated (or actual) number of rows in the table. However, you may choose to supply this information yourself.
- Index names must be unique for all indexes defined on a table
- An index must be in the same page group as the table on which it is defined.

Specifying Physical Attributes

When creating an index, you can specify physical attributes that are normally generated automatically. Specifying explicit values for this information allows you to create and maintain tables that have identical physical attributes and can therefore be accessed through a single schema definition.

Example

In this example, an index has been created on the employee table. The keys in the index are LAST_NAME, FIRST_NAME. The index does not require that the last name/first name combination be unique. The index will be located physically in a separate area from the data in the table.

```
create index em_name_ndx on prod.employee (last_name, first_name)
    in emp.empreg1;
```

Defining a Referential Constraint

You create a referential constraint by issuing the CREATE CONSTRAINT statement and specifying the referenced and referencing tables and columns.

Things You Can Specify

1. Name of the constraint
2. The *referencing* table and column(s)
3. The *referenced* table and column(s)
4. Whether the referential constraint is *linked* or *unlinked* (the default)
5. Options such as clustered (ORDER BY) or indexed (INDEX)

Considerations

- The referenced column values of each row in the referenced table must be unique in the database. Therefore, ensure that either a unique CALC key or a unique index key is defined on the referenced columns.
- The datatype of a referencing column must be the same as its referenced column.
- If you specify an *unlinked* referential constraint:
  - The referencing table must have a CALC key or index defined on the referencing columns.
  - The order of the columns must be the same as the unique CALC key or index on the referenced columns.
  - If using an index on the referencing columns, the index can contain columns in addition to the referencing columns. The referencing columns must precede any additional columns in the index key.
- If you are defining a self-referencing constraint, it must be *unlinked*.
- Referential constraints (linked and unlinked) may not cross page group boundaries, meaning that the areas in which the referenced and referencing tables reside must have the same page group.

Example - Linked Referential Constraint

In this example, a linked referential constraint has been created to ensure that the employee ID in the benefits table is a valid ID by checking it against the employee IDs in the employee table. The referential constraint is indexed and ordered by the fiscal year.

```
create constraint emp_benefits
   benefits (emp_id)
   references employee (emp_id)
   linked index
     order by (fiscal_year desc);
```

Example - Unlinked Referential Constraint

In this example, an unlinked referential constraint has been created to ensure that an employee's manager is a valid employee. Since this is a self-referencing constraint (both columns being in the same table), it must be unlinked. UNLINKED is the default.

```
create constraint manager_emp
   employee (manager_id)
   references employee(emp_id);
```

Dropping a Default Index

You add or drop a default index by issuing an ALTER TABLE statement on the table whose default index you want to drop.

Things You Can Specify

- Use the ADD DEFAULT INDEX parameter to create a default index for the named table.

  Note: The table must not already have a default index associated with it.

- Use the CASCADE parameter to drop all indexes in which the named column is an indexed column following entities.

  Note: If CASCADE is not specified, the column must not participate in a referential constraint or index, or be named in a view.

Considerations

It may not always be appropriate to drop a default index.

Note: See the *Database Design Guide* document for complete information about retaining or dropping default indexes, and see the *SQL Reference Guide* for complete information on ALTER TABLE.

Example

In the following example, the default index is dropped from the EMPLOYEE table .

```
alter table prod.employee
      drop default index;
```

Creating a View

You create a view by issuing the CREATE VIEW statement and specifying the view column names, the table(s) and column(s) from which the view is derived, and data restrictions, if any.

Things You Can Specify

1. Name of a view, using a schema qualifier unless you have specified a default schema name in the SET SESSION statement

   Note: For more information about session management statements, see the *CA IDMS SQL Reference Guide*.

2. A column list if there are computations or duplicate column names in the result table of the view definition
3. An appropriate SQL select statement

   Note: For a complete discussion of SQL select statements, see the *CA IDMS SQL Reference Guide*.

4. A check option to ensure that only data values that satisfy the SELECT statement are inserted or updated through the view.
5. Physical attributes, including a synchronization stamp.

Specifying Physical Attributes

When creating a view, you can specify physical attributes that are normally generated automatically. Specifying explicit values for this information allows you to create and maintain views that have identical attributes and can therefore be accessed through a single schema definition.

Care should be exercised when specifying a specific timestamp, since its purpose is to enable the detection of discrepancies between a view and its definition. If explicitly specified, the timestamp should always be set to a new value following a definitional change so that the change is detectable to the run time system.

Considerations

- You cannot define an index on a view
- Updatable views are syntactically valid anywhere in SQL DML statements that tables are; a view is updatable when the SELECT statement references only one table and when the view projects no computed values
- If the WHERE clause of the SELECT statement contains a subquery, you cannot use the check option
- Avoid using an asterisk (*) in the SELECT statement of your view. If a column is added to the underlying table, the view becomes invalid and must be dropped and recreated.

Example - Single Table View

In the following example, a simple view is defined on the EMPLOYEE table.

```
create view prod.emp_home_phone
   as select emp_id, emp_lname, emp_fname, phone
      from prod.employee;
```

Example - Updatable View

In the following example, a view is defined with the check option to restrict rows that can be updated and inserted. Using the view, the value of DEPT_ID cannot be changed to something other than 'SALES', and new rows must have a DEPT_ID of 'SALES'.

```
create view hr.sales_employee
   as select emp_id, emp_lname, emp_fname, dept_id, emp_ssno
      from prod.employee
      where dept_id = 'SALES'
   with check option;
```

Example - Nonupdatable View

In the following example, a view is defined with three columns derived from two tables. Since the third column includes both aggregate functions and an arithmetic operation, the CREATE VIEW statement must specify names for the columns in the view.

This view is nonupdatable because the SELECT references more than one table and because the view projects computed values.

```
create view prod.emp_vacation
   (emp_id, dept_id, vac_time)
   as select e.emp_id, dept_id, sum(vac_accrued) - sum(vac_taken)
      from prod.employee e, prod.benefits b
      where e.emp_id = b.emp_id
      group by dept_id, e.emp_id;
```

Note: For more information about SQL syntax, see the *CA IDMS SQL Reference Guide*.

Chapter 9: Defining a Database Using Non-SQL

This section contains the following topics:

Overview

This chapter provides information about Step 4, defining the logical components (schema, subschema) of the database.

Steps to Define a Database

To use non-SQL methods to define your database, follow these steps:

1. Design and size the database using information provided in the *CA IDMS Database Design Guide* document.

2. Define in the system dictionary the segments that represent the physical database. Include the segments in your DMCL, and generate, punch, and link edit the DMCL. For more information on the physical database, see Chapter 4, “Defining Segments, Files, and Areas".

   Note: You can defer this step until after you define the schema and subschema.

3. Allocate and format the operating system files as described in Chapter 17, "Allocating and Formatting Files”.

   Note: You can defer this step until after you define the schema and subschema.

4. Define a schema and one or more subschemas.

Schemas and Subschemas

CA IDMS/DB needs descriptions of databases to manage those databases. To satisfy this requirement, the database administrator defines two logical components of the non-SQL database:

Schema

The schema is a *complete* description of a database, including the names and descriptions of all areas, records, elements, and sets. The major purpose of the schema is to provide definitions from which to generate subschemas.

Subschema

A subschema provides a view of the database as seen by an application program. This view is often a subset of the complete schema definition. A subschema is used at runtime to provide the DBMS with a description of those portions of the database that are accessible to the application program.

The subschema can restrict access to the database in the following ways:

- The subschema identifies the areas, records, elements, and sets which are accessible.
- The subschema identifies the Data Manipulation Language (DML) functions which can be performed.

Subschemas also allow you to define logical records. Logical records are a view of one or more base records and a set of operations performed on those records.

Other entities defined within the process of schema and subschema definition are records, sets, areas, indexes, and CALC keys.

Note: For a complete discussion of non-SQL database components and how to decide which components and options you will use in your database, see the *CA IDMS Database Design Guide*.

Storing Schema and Subschema Source

Source descriptions for schemas and subschemas are kept in the DDLDML area of the dictionary.

Many software components need database descriptions that are not in object form. For example, DML compilers need a source from which they can generate record descriptions within user-written programs; the IDMSRPTS utility needs a source from which it can produce database reports, and so on. Source descriptions provide a form that is readable by the software when performing these non-DBMS functions.

Load Modules are Maintained for Subschemas

Load modules are maintained for subschemas. Subschema load modules are kept in the DDLDCLOD area of the dictionary and, optionally, in a load library.

Load modules consist of machine-readable code that CA IDMS/DB uses at runtime to transfer data between the program and the database.

Schema and Subschema Compilers

Schema Compiler

The schema compiler, IDMSCHEM, performs the following functions:

- Accepts as input DDL statements that describe the areas, records, elements, and sets of the database
- Evaluates the syntax and logic of the input
- Places source descriptions of the schema and its components into the dictionary
- Produces a list of the compiler's activities

Subschema Compiler

The subschema compiler, IDMSUBSC, performs the following functions:

- Accepts as input DDL statements that describe the subschema as follows:
  - Identifies selected areas, records, elements, and sets of the database
  - Defines logical records
  - Places restrictions on allowable DML verbs
- Validates the syntax and logic of the input
- Places a source description of the subschema into the dictionary
- Generates a subschema load module and places it into the dictionary
- Produces a list of the compiler's activities

You can define any number of subschemas for each schema. One subschema might include all areas, records, and sets in the schema while another might contain only those areas, records, and sets needed for a program accessing the database. Usually you define one subschema for each group of similar applications that access the database.

Additional Functions of the Compilers

In addition to the functions stated above, SCHEMA and SUBSCHEMA statements can:

- Add, modify, delete, display, or punch a schema or subschema description
- Secure the schema or subschema definition
- Authorize users to issue specific verbs against the schema or subschema definition

Note: For more information about using the schema and subschema compilers, see Chapter 10, "Using the Schema and Subschema Compilers".

Defining a Schema

Order of Schema Component Definition

When you add a new schema to the dictionary, you must submit the ADD SCHEMA statement first. Although you can add most statements in any order, cross references to nonexistent components generate error messages. To avoid error messages, submit statements in this order:

1. SCHEMA statement
2. AREA statements
3. RECORD statements (and associated ELEMENT substatements)
4. SET statements
5. VALIDATE statement

Note: If VALIDATE is not executed successfully, the schema cannot be used by other software components. (Subschemas cannot be defined and utilities that require the schema name as input cannot be executed.)

SCHEMA Statement

The SCHEMA statement performs the following:

- Identifies the schema
- Secures the schema definition
- Establishes schema currency

When you issue an ADD SCHEMA statement, a new schema description is created in the dictionary. Default values established through the SET OPTIONS statement (see 11.5, “SET OPTIONS Statement”) can be used to supplement the user-supplied description.

ADD also sets the schema's status to IN ERROR. A VALIDATE statement must set the status to VALID before a subschema or CA IDMS/DB utility can reference the schema.

Procedure

1. Name the schema
2. Optionally add descriptive information
3. Optionally specify automatic record ID assignment
4. Optionally identify the schema that this schema is derived from
5. Optionally provide security information
6. Optionally provide comments and user-defined attributes

Examples

The following example shows the minimum SCHEMA statement required to establish a database.

```
add schema name is sampschm.
```

The following example shows a complete SCHEMA statement.

```
add schema name is empschm version is 1
    assign record ids from 3000
    derived from schema oldschm version is 1
    include user is kla registered for all
    public access is allowed for display
    include status is production
    comments 'this schema is based on a former employee schema'
            -'used before the addition of the new divisions'.
```

AREA Statements

AREA statements identify an area of the database. Depending on the verb and options coded, the AREA statement can also:

- Add, modify, delete, display, or punch the area description
- Determine which (if any) database procedures will be executed when the area is accessed at runtime

The schema compiler applies AREA statements to the current schema. See 9.7, “Establishing Schema and Subschema Currency”.

The ADD AREA statement causes CA IDMS/DB to create a new area description in the dictionary and associates it with the current schema.

Procedure

1. Name the area
2. Optionally specify database procedures to be called

Note: You can copy an area description from another schema

Example

The following example shows an AREA statement including calls to database procedures:

```
add area name is org-demo-region
    call secdbproc before ready for exclusive update
    call chkdbproc before rollback.
```

SAME AS

SAME AS copies an entire area description including database procedure information from an area in another schema into the current schema. The SAME AS clause must precede all other optional clauses.

RECORD Statements

RECORD statements identify a non-SQL database record type. Depending on the verb, options, and substatements coded, the RECORD statements can also:

- Add, modify, delete, display, or punch the record description
- Assign the record type to an area
- Determine which (if any) database procedures will be executed when occurrences of the record type are accessed at runtime
- Create a dictionary description of the record, including its synonyms, elements, and element synonyms or associate the record with an existing structure

The schema compiler applies RECORD statements to the current schema.

The ADD RECORD statement creates a new schema record description in the dictionary and associates it with the current schema.

Unless the SHARE clause is used, ADD RECORD creates a record structure for the schema record. The record structure's name is the same as that of the schema record. The structure is automatically assigned a version number, which distinguishes the record from others that have the same name in the dictionary. The schema compiler uses NEXT HIGHEST when assigning record version numbers.

Note: It is better to use the SHARE clause rather than define the record structure in the schema. The SHARE clause allows you to maintain control of the record versions stored in the dictionary.

SHARE

The SHARE STRUCTURE and SHARE DESCRIPTION clauses allow the schema to share the structure of either a dictionary record (IDD record) or a record that belongs to another schema.

The SHARE clause connects an existing record structure to the schema record. The schema record shares the dictionary description of an existing record, including its synonyms, elements, and element synonyms. The SHARE clause does not create a *new* record structure.

Note the following considerations about using SHARE:

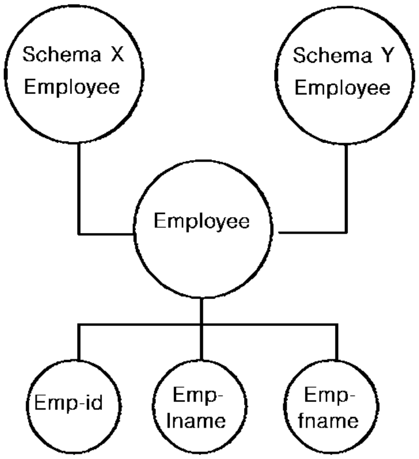
- All schema records that share a single structure must have the same name
- Any number of schema records can share a single structure
- The structure is shared equally among the records; no single schema owns the structure
- The SHARE clause must precede any RECORD SYNONYM clauses. Synonyms are assigned to the structure and are therefore available to all schema records that share the structure.
- The schema compiler does not allow modification of a shared structure except to include record synonyms. Nonstructural information (record ID, location mode, and so on) is maintained separately for each schema record and can be modified.
- The SHARE clause and ELEMENT substatements (14.5, “Element Substatement”) are mutually exclusive. Use SHARE to connect the record to an existing structure; use ELEMENT substatements to create a new structure for the schema record.

  Do not use ELEMENT substatements for any schema record that shares a structure. Once SHAREd, a schema record should always be maintained through SHARE clauses.

Both SHARE STRUCTURE and SHARE DESCRIPTION cause the schema record to share the structure of an existing record.

Two Schemas Sharing One Record Structure

The following diagram shows two schemas sharing the structure of the EMPLOYEE record.



SHARE STRUCTURE

When using SHARE STRUCTURE, you must supply the appropriate:

- Record ID
- Location mode
- VSAM type
- Area association
- Minimum root
- Minimum fragment
- CALL clauses

Example

The following example shows a RECORD statement for SKILL which shares the structure of the SKILL record in the schema OTHRSCHM.

```
add record name is skill
  share structure of record skill
     of schema othrschm
  location mode is calc using skill-code
     duplicates are not allowed
  within area org-demo-region
  minimum root length is control length
  minimum fragment length is record length
  call idmscomp before store
  call idmscomp before modify
  call idmsdcom after get.
```

SHARE DESCRIPTION

SHARE DESCRIPTION allows the schema record to share the structure of a record that belongs to another schema. Unlike SHARE STRUCTURE, SHARE DESCRIPTION copies the *entire* record description (record ID, location mode, etc.) from the owning schema to the schema record named as the object of the ADD statement. You do not have to add anything.

Example

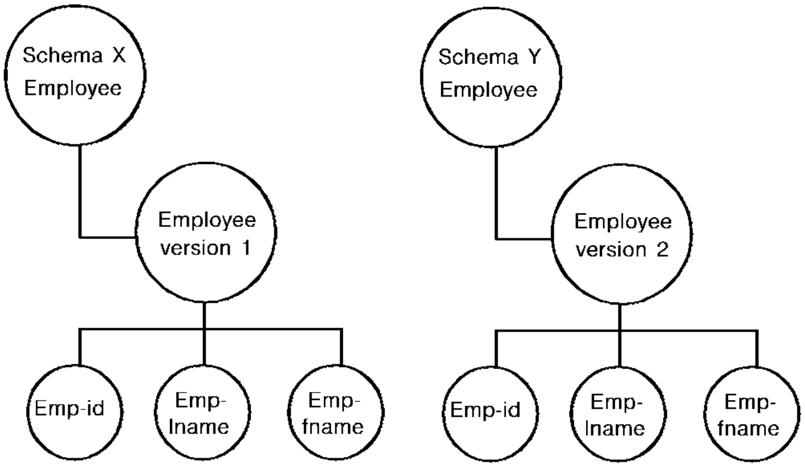
In the following example, the SKILL record in the current schema shares the structure of the SKILL record in EMPSCHM (version 1). Each record has its own copy of nonstructural information.

```
add record name is skill
  share description of record skill
    of schema empschm version 1.
```

COPY ELEMENTS

The COPY ELEMENTS substatement uses the structure of an existing record type to generate new element descriptions for the record type. (The SHARE clause of the RECORD statement does not generate new element descriptions; it uses existing ones.)

Separate Record Structures with Identical Elements



The COPY ELEMENTS substatement requests that all elements from a record description already stored in the dictionary be included in the new record structure. The record description may have been stored through another schema or through the IDD DDDL compiler. COPY ELEMENTS can be used in place of ELEMENT substatements (see below) to define all of the record's elements or only some of them. When COPY ELEMENTS supplies some of the record's elements, use ELEMENT substatements to supply the rest.

SHARE and COPY ELEMENTS

The differences between SHARE STRUCTURE, SHARE DESCRIPTION, and COPY ELEMENTS are as follows:

SHARE DESCRIPTION	SHARE STRUCTURE	COPY ELEMENTS
Shares the structure of another schema record	Shares the structure of either a dictionary record (IDD record) *or* another schema record	Creates new element descriptions based on existing record structures
Uses existing element descriptions	Uses existing element descriptions	Creates new element descriptions

SHARE DESCRIPTION	SHARE STRUCTURE	COPY ELEMENTS
Copies the *nonstructural* part of the existing schema record: ■ Record ID ■ Location mode ■ VSAM type ■ Area ■ Minimum root length ■ Minimum fragment length ■ Database procedures	Does not copy nonstructural information	Does not copy any record information

ELEMENT Substatements

The ELEMENT substatements identify the element of a schema record. Because elements cannot exist in a database except as components of a record, schema elements are considered subordinate to schema records. Consequently, all ELEMENT substatements for a single record must immediately follow the RECORD statement in a single execution of the schema compiler.

The ELEMENT substatement uses COBOL-like syntax to describe elements. Additional clauses provide CA IDMS/DB-specific information and documentational entries.

The ELEMENT substatement associates an element with the record and, if the element does not already exist, adds the element description to the dictionary. The element descriptions cannot be modified individually or deleted using these substatements. To change element descriptions, modify the record description and *respecify* all of the record's elements.

The minimum ELEMENT substatement required for the element to be a valid schema component depends on whether the element is a group or elementary item:

Item	Required
Group item	■ Level ■ Name
Elementary item	■ Level number ■ Name ■ Picture (or usage)

Example

Minimal ELEMENT substatements are shown below:

```
02 claim-date.
   03 claim-year   pic 99.
   03 claim-month  pic 99.
   03 claim-day    pic 99.
```

Mixing ELEMENT and COPY ELEMENTS Substatements

You can mix ELEMENT and COPY ELEMENTS substatements in any sequence necessary to describe the structure of the record. However, because the level number of copied elements are the same as those in the base record, you should take care in mixing elements of different levels. To mix ELEMENT and COPY ELEMENTS substatements and to change the level numbers within the record, follow these steps:

1. Code ELEMENT and COPY ELEMENTS substatements to put the elements into their appropriate positions in the record structure
2. Online, issue a DISPLAY RECORD with AS SYNTAX and VERB MODIFY for the record; in batch mode, code PUNCH instead of DISPLAY.
3. Change the affected level numbers only. *Do not erase unaffected elements*; all elements for a single record must always be presented together.
4. Submit the new statement to the compiler

Example

In the following example, the structure of NEW-COVERAGE is generated by copying elements from the COVERAGE record and the IDD-built CARRIER-DETAIL record, and by coding new element descriptions in line:

```
add record name is new-coverage
  location mode is via emp-coverage set
  within emp-demo-region area
  copy elements from record coverage
    of schema empschm version 1.
  02 cov-carried-id   pic 99.
  02 cov-carrier-name pic x(20).
  copy elements from record carrier-detail.
```

The result of the above activity is as follows:

```
01 new-coverage
   02 cov-select-date.
      03 cov-select-year    pic 99
      03 cov-select-month   pic 99
      03 cov-select-day     pic 99
   02 cov-termin-date.
      03 cov-termin-year    pic 99
      03 cov-termin-month   pic 99
      03 cov-termin-day     pic 99
   02 cov-type              pic x.
   02 cov-insplan-code      pic xxx.
   02 cov-carrier-id        pic 99.
   02 cov-carrier-name      pic x(20).
   02 cov-carr-no-of-claims pic 99 comp.
   02 cov-carr-claims-processed
                        occurs 0 to 100
                        depending on
                        cov-carr-no-of-claims
      03 cov-carr-payment   pic x.
         88 prompt          value '9'.
         88 over-30-days    value '4'.
         88 over-60-days    value '1'.
      03 cov-carr-courtesy  pic x.
      03 cov-carr-check     pic x.
         88 cleared         value 'C'.
         88 bounced         value 'B'.
```

Procedure

1. Name the record
2. Identify where the structure of the record is to come from:
   - Structure shared with an existing record structure
   - Structure defined in this schema
3. Optionally specify the record ID
4. Specify the location mode for the record
5. Specify the area where record occurrences will be stored; optionally specify a subarea
6. Optionally specify minimum root and fragment information for variable length records
7. Optionally specify database procedures to be called

Note: If you specified in the SCHEMA statement that record IDs were to be set up automatically, you can still override the ID in the RECORD statement.

Example

The following example defines a schema record which has the same description as another record in schema DEMOSCHM. The employee record will be stored CALC based on the EMP-ID element with a portion of the EMP-DEMO-REGION area. The portion of the area is defined with the SUBAREA clause. The subarea name is actually defined in the DMCL and resolved at runtime.

```
add record name is employee
   share structure of record emp version is 10 of schema demoschm
   location mode is calc using (emp-id) duplicates are not allowed
   within area emp-demo-region
         subarea low-pages
   call idmscomp before store
   call idmscomp before modify
   call idmsdcom after get.
```

SET Statements

The SET statements identify and describe a set. Depending on the verb, the SET statements can add, modify, delete, display, or punch the set description (see 14.7, “SET Statement").

Note that if a set's *owner record* is deleted, the set is automatically deleted. Additionally, the deleted record and set are deleted from all subschema descriptions associated with the current schema. However, if the *member record* is deleted, the set remains. To delete the set (if it has no other member records), use the DELETE SET statement.

The schema compiler applies SET statements to the current schema.

The ADD SET statement creates a new set description in the dictionary and associates it with the current schema.

Procedure

1. Name the set
2. Specify order
3. Specify the mode
4. Specify owner and members
5. Specify set options

Note:

- If you intend to have prior pointers, do not forget to specify MODE IS CHAIN LINKED TO PRIOR.
- If you are creating a system-owned index, the owner is SYSTEM.

Example

The following example shows a SET statement.

```
add set name is insplan-rider
  order is last
  mode is chain
  owner is insplan
  member is rider
  mandatory automatic.
```

SAME AS

The SAME AS clause copies an entire set description including order, mode, owner, and members from a set in another schema into the current schema. SAME AS must precede all other optional clauses.

VALIDATE

Schema Status

CA IDMS/DB requires that a valid schema reside in the dictionary before any other activity involving the database can begin. Each schema in the dictionary carries a status of either IN ERROR or VALID as follows:

Status	Indicates...	Status set by...
IN ERROR	The schema was not processed by an error-free VALIDATE statement and prevents other CA IDMS/DB software (subschema compiler and utilities) from using the schema	After the execution of an ADD SCHEMA or MODIFY SCHEMA statement

Status	Indicates...	Status set by...
VALID	The schema is usable by other CA IDMS/DB software	After error-free execution of the VALIDATE statement

Only the schema compiler updates the status.

Verification

VALIDATE causes the schema compiler to verify the relationships among all components of the schema that is current for update. Based on this verification, the schema compiler takes one of the following actions:

Result	Compiler action
No errors found	Compiler sets schema status to VALID
Errors found	Compiler issues messages indicating the exact nature of each error

Other Results of VALIDATE

In addition to the verification, VALIDATE causes the schema compiler to resolve pointer positions for which AUTO was specified in set description statements.

The VALIDATE statement can be used at any time to verify the relationships of schema components. For example, if you have not yet defined sets, but want to verify the schema's record structures, you can use VALIDATE. In this case, however, you should anticipate a warning for those records whose location mode is VIA an undefined set.

Procedure

Issue the VALIDATE statement:

```
validate.
```

Defining a Subschema

The subschema copies its logical database definitions from the schema. You must define a valid schema and store it in the dictionary before you can create a subschema.

Order of Subschema Component Definition

When you add a new subschema to the dictionary, you must submit the ADD SUBSCHEMA statement first. Although you can add most statements in any order, cross references to nonexistent components generate error messages. To avoid error messages, submit statements in this order:

- SUBSCHEMA statement
- AREA statements
- RECORD statements
- SET statements
- LOGICAL RECORD statements
- PATH-GROUP statements
- VALIDATE statement
- GENERATE statement

Subschema Statement

What It Does

The SUBSCHEMA statement:

- Identifies the subschema
- Associates it with a schema
- Secures the subschema definition
- Establishes subschema currency

Once a specific subschema becomes current, the subschema compiler applies subsequent statements to that subschema.

Procedure

1. Name the subschema
2. Name the schema from which this subschema is derived
3. Optionally provide a description
4. Specify the usage
5. Optionally include security information
6. Optionally include comments

Note: Be explicit about the usage mode for your subschema. Specify either LR or DML; only in cases where both LRF and DML are used should you specify MIXED (for more information, see the *CA IDMS Logical Record Facility Guide* document).

Example

The following example shows the definition of the subschema EMPSS01.

```
add subschema name is empss01
    of schema name is empschm version is 1
    description is 'subschema for adding/modifying employees'
    public access is allowed for all
    usage is lr.
```

AREA Statements

AREA statements identify areas to be included in this subschema. The area descriptions are copied from the schema area descriptions. Depending on the verb and options coded, the AREA statements can also:

- Determine the usage modes in which programs using the subschema can ready the area
- Determine the default usage mode for programs that do not issue READY statements
- Modify, delete, display, or punch a subschema area

The subschema compiler applies AREA statements to the current subschema.

Procedure

1. Name the area
2. Optionally specify usage modes that are not allowed
3. Optionally specify default usage mode for the area

Note: The default for usage modes is that the mode is allowed. Specify those usage modes you do *not* want allowed.

Example

The following example shows the definition of the area ORG-DEMO-REGION being copied into the current subschema.

```
add area org-demo-region
  exclusive update is not allowed
  default usage is shared update.
```

RECORD Statements

RECORD statements identify records to be included in this subschema. The record descriptions are copies from the schema descriptions. Depending on the verb and options coded, the RECORD statements can also:

- Specify which record elements can be accessed through the subschema
- Specify which DML verbs can be issued against the record
- Specify the order in which record descriptions occur within the subschema
- Modify, delete, display, or punch a subschema record description

The subschema compiler applies RECORD statements to the current subschema.

Procedure

1. Name the record
2. Optionally identify the elements that can be accessed through the subschema
3. Specify which DML verbs will not be allowed

Note:

- A simple ADD RECORD statement copies a record in its entirety includingall its elements from the schema description into the subschema definition.
- You can change the order of the elements from that specified in the schema.
- You can add additional security and control by restricting the DML commands that programs using this subschema can issue against each record.

Example

The following example shows the definition of the record SKILL being copied into the current subschema.

```
add record skill
  store is not allowed
  erase is not allowed.
```

SET Statements

SET statements identify sets to be included in this subschema. The set description is copied from the schema description. Depending on the verb, the SET statements can also:

- Determine which DML verbs can be issued against the set
- Modify, delete, display, or punch a subschema set description

The subschema compiler applies SET statements to the current subschema.

Procedure

1. Name the set
2. Optionally specify which DML verbs will not be allowed

Note:

- If the set's *owner record* is deleted, either from theschema or from the subschema, the set is automatically deleted from the subschema.
- If the set's *member record* is deleted, either from the schema or from the subschema, the set remains in the subschema.
- If a set is added to the subschema, the owner of the set must also be added to the subschema
- If one or more sets associated with a record is not included in the subschema, certain update operations on the record are prohibited, as follows:
  - If a set in which the record is an owner is missing, the record cannot be erased
  - If a set in which the record is a member is missing, the record cannot be erased and:
    - If the set has a membership of AUTOMATIC, the record cannot be stored
    - If the set is sorted, the record cannot be modified

Example

The following example shows the definition of the set SKILL-EXPERTISE being copied into the current subschema.

```
add set name is skill-expertise.
```

LOGICAL RECORD Statements

LOGICAL RECORD statements define a logical record that programs using the subschema can access.

A logical record is defined by naming the logical record and all the subschema records that participate in it; these subschema records are known as *logical-record elements*. The records must participate in the subschema (through ADD RECORD statements) before they can be named as logical record elements in the LOGICAL RECORD statement.

When a DML processor copies a logical-record description into a program, each logical-record element is subordinate to the logical record itself. The sequence of logical-record elements in the copied description is the same as that in DDL LOGICAL RECORD statement. If a subschema record occurs more than once in a single logical record, the additional occurrences must be assigned unique identifiers called *roles*.

The subschema compiler applies LOGICAL RECORD statements to the current subschema.

Note: For more information about creating logical records, refer to the *CA IDMS Logical Record Facility Guide* document.

Procedure

1. Name the logical record
2. Name the records that are components of this logical record
3. Optionally specify error information
4. Optionally include comments

Example

The following example shows the definition of the logical records MANAGER-STAFF and DEPT-ROSTER.

```
add lr name is manager-staff
  elements are employee
               structure
               employee role name is staff.
add lr name is dept-roster
  elements are department
               employee role name is staff.
```

PATH-GROUP Statements

PATH-GROUP statements define paths for a specific logical record. At runtime, LRF services program requests by following one of the paths to access the logical record.

For each logical record, at least one path group, and at most four (one for each DML verb that can be used to access the logical record), must be defined. A path group can contain any number of paths. Which path LRF uses at runtime is determined by selection criteria, both in the path group and in the program requesting LRF services.

Note: For more information about logical records and path groups, see the *CA IDMS Logical Record Facility Guide* document.

The subschema compiler applies PATH-GROUP statements to the current subschema.

Procedure

1. Name the type of path group
2. Add appropriate DML statements

Example

```
add path-group name is store emp-pers-data
  select
    find first department
       where calckey eq dept-id-0410 of lr
       on 0326 return no-dept
       on 0000 next
    find first office
       where calckey eq office-code-0450 of lr
       on 0326 return no-office
       on 0000 next
    find first employee
       where calckey eq emp-id-0415 of lr
       on 0000 return emp-exists
       on 0326 next
    store employee
       on 0000 next
```

Subschema Validation and Generation

After you describe the subschema, the dictionary contains the subschema description, but no subschema load module yet exists in the load area of the dictionary. A load module can be generated only from a *valid* subschema description.

Subschema Status

Each subschema description in the dictionary carries a status of either IN ERROR or VALID as follows:

Status	Indicates...	Status set by...
IN ERROR	The subschema has been added or modified but has not been validated. This status prevents the generation of a load module for the subschema.	■ An ADD SUBSCHEMA or MODIFY SUBSCHEMA statement ■ Any schema modification that affects the subschema (for example, deletion of a set)
VALID	The subschema has been validated and load modules can be generated	■ The error-free execution of a VALIDATE or GENERATEstatement ■ The error-free execution of a schema compiler REGENERATE statement.

You can validate the subschema and generate the load module in a single step (using the GENERATE statement) or you can validate the subschema at any time without generating a load module (using the VALIDATE statement).

VALIDATE

The VALIDATE statement instructs the subschema compiler to verify the relationships among all components of the subschema. Based on this verification, the compiler takes one of the following actions:

Result	Compiler action
No errors found	Compiler sets subschema status to VALID
Errors found	Compiler issues messages indicating the exact nature of each error

You usually use VALIDATE for dry runs of the subschema compiler since it causes the compiler to check the components but not to create subschema load modules.

Procedure

Issue the VALIDATE statement:

```
validate.
```

GENERATE

The GENERATE statement instructs the compiler to create subschema tables for the subschema that is current and to store them as a load module in the dictionary load area. For GENERATE to produce the new subschema load module, the current subschema must be valid. So, if a VALIDATE statement has not been specified for the subschema, the GENERATE statement causes the compiler to perform validation before creating the subschema tables.

Procedure

1. Issue the GENERATE statement, as follows:

   ```
   generate.
   ```

Security Checking

The schema and subschema compilers maintain security to ensure that no unauthorized person uses the compilers to perform secured operations. The compilers perform security checking operations when:

- The verb is SIGNON, VALIDATE, or GENERATE
- The SET OPTIONS statement contains REGISTRATION OVERRIDE
- The component type is SCHEMA
- The component type is SUBSCHEMA
- The statement is the first statement of the session

In any of the above cases, the compiler determines whether the requested operation is secured. If the operation is not secured, the compiler bypasses the security check and begins processing the statement. If the operation is secured, the compiler checks the user's description in the dictionary to determine whether the user is authorized to perform an operation. If the user is authorized, the compiler processes the input statement; if not, the compiler issues an error message.

Types of Security Checked

The compilers check four kinds of security:

- Compiler security
- Registration override security
- Verb security
- Component security

Each kind of security is presented separately below; each topic includes the following kinds of information:

- When security is checked
- How security is turned on or off
- How the compiler determines who the issuing user is
- What constitutes an authorized user

Checking Compiler Security

The schema and subschema compilers check compiler security:

- When SIGNON is issued
- When the first statement of the session is issued (implicit SIGNON)

Compiler security is turned on or off through the IDD DDDL statement, SET OPTIONS FOR DICTIONARY SECURITY FOR IDMS IS ON/OFF.

Note: This IDD DDDL statement also turns verb security on or off; compiler and verb security cannot be set independently.

Determining Who is Issuing the Statement

To determine who is issuing the statement, the compiler looks at the user name specified in the SIGNON statement. If the SIGNON statement is not issued or does not include the USER clause, the user name defaults as described in the SET OPTIONS presentation under Chapter 10, “Using the Schema and Subschema Compilers”.

An authorized user, for this function, is one whose description in the dictionary includes authority to use the compiler. Compiler authority is assigned through one of the following IDD DDDL USER statements (use MODIFY for existing user descriptions):

Statement	Action
ADD USER NAME IS *user-name* AUTHORITY FOR *any verb* IS ALL.	Assigns authority to use both compilers
ADD USER NAME IS *user-name* AUTHORITY FOR *any verb* IS IDMS.	Assigns authority to use both compilers
ADD USER NAME IS *user-name* AUTHORITY FOR *any verb* IS SCHEMA.	Assigns authority to use the schema compiler only
ADD USER NAME IS *user-name* AUTHORITY FOR *any verb* IS SUBSCHEMA.	Assigns authority to use the subschema compiler only

Checking Registration Override Security

The schema and subschema compilers check registration override security when they encounter a SET OPTIONS statement containing a REGISTRATION OVERRIDE clause.

Unlike the other kinds of security, this one cannot be turned on or off; that is, the compiler always checks for an authorized user when it encounters a REGISTRATION OVERRIDE clause.

Determining Who is Issuing the Statement

To determine who is issuing the REGISTRATION OVERRIDE clause, the compiler looks at the PREPARED BY and REVISED BY user names in the SET OPTIONS statement. If the SET OPTIONS statement does not include either clause, or if user signon override is not allowed, the user name defaults as described in the SET OPTIONS presentation under Chapter 10, “Using the Schema and Subschema Compilers”.

An authorized user for the REGISTRATION OVERRIDE clause is one whose description in the dictionary includes all authorities. All authorities are assigned through the following IDD DDDL USER statement (use MODIFY for existing user descriptions):

```
ADD USER NAME IS user-name
  AUTHORITY IS ALL.
```

Checking Verb Security

The schema and subschema compilers check verb security whenever a SCHEMA statement (schema compiler only) or SUBSCHEMA statement (subschema compiler only) is issued. Note that verb security is not checked for each component of a schema or subschema. Once a user passes security for a schema or a subschema, all of its components are available to the user.

Turning Verb Security On or Off

Verb security is turned on or off through the IDD DDDL statement, SET OPTIONS FOR DICTIONARY SECURITY FOR IDMS IS ON/OFF.

Note: This IDD DDDL statement also turns compiler security on or off; verb security and compiler security cannot be set independently.

Determining Who is Issuing the Statement

To determine who is issuing the SCHEMA or SUBSCHEMA statement, the compiler looks at four areas; if any area contains the name of an authorized user, security is satisfied and the compiler processes the request:

- The SCHEMA or SUBSCHEMA statement PREPARED BY clause
- The SCHEMA or SUBSCHEMA statement REVISED BY clause
- The current session option for PREPARED BY
- The current session option for REVISED BY

Note: If user signon override is not allowed, the user issuing the statement is *always* assumed to be the user known to the execution environment. PREPARED BY and REVISED BY user specifications are ignored.

An authorized user, for this function, is one whose description in the dictionary includes authority to issue the verb specified in the SCHEMA or SUBSCHEMA statement, *in conjunction with* the authority to use the compiler. Verb authority is assigned through IDD DDDL USER statements, such as those in the following examples:

```
ADD USER NAME IS KCO                 assigns authority to use all
    AUTHORITY FOR UPDATE             verbs in each DDL compiler
      IS IDMS.

ADD USER NAME IS BAC                 assigns authority to use MODIFY,
    AUTHORITY FOR MODIFY             DISPLAY, and PUNCH in each DDL
      IS IDMS.                       compiler

ADD USER NAME IS TWG                 assigns authority to use DELETE,
    AUTHORITY FOR DELETE             DISPLAY, and PUNCH in the schema
      IS SCHEMA.                     compiler only

ADD USER NAME IS JFD                 assigns authority to use DISPLAY
    AUTHORITY FOR DISPLAY            and PUNCH in the schema compiler
      IS SCHEMA.                     only
```

While schema authority only allows the user to access the schema compiler, any subschema updates resulting from authorized schema updates are allowed (for example, deleting a set from the schema causes the set to be deleted from the subschemas associated with that schema).

Note: For more information about assigning verb authority, see the *CA IDMS IDD DDDL Reference Guide*.

Checking Component Security

The schema compiler checks the security of a specific schema whenever a SCHEMA statement (other than ADD SCHEMA) is issued for that schema; the subschema compiler checks security of a specific subschema whenever a SUBSCHEMA statement (other than ADD SUBSCHEMA) is issued for that subschema. Note that this security is not checked for each component of a schema or subschema. Once a user passes security for a schema or a subschema, all of its components are available to the user. Component security applies to every existing schema and subschema, regardless of whether compiler security is on.

PUBLIC ACCESS Clause

Security for a specific schema or subschema is set through the PUBLIC ACCESS clause of the SCHEMA or SUBSCHEMA statement. A schema or subschema is said to be unsecured if PUBLIC ACCESS IS ALLOWED FOR ALL is in effect; any other public access specification places some level of security on the schema or subschema.

Examples

The following examples show how component security is set:

```
MOD SCHEMA EMPSCHM                      turns off security for EMPSCHM
    PUBLIC ACCESS IS ALLOWED
      FOR ALL.

MOD SUBSCHEMA EMPSS01                   turns on security for all verbs
    OF SCHEMA EMPSCHM                   issued against EMPSS01
    USER IS JFD
      REGISTERED FOR ALL
    PUBLIC ACCESS IS ALLOWED
      FOR NONE.

MOD SUBSCHEMA EMPSS02                   turns off security for DISPLAY
    OF SCHEMA EMPSCHM                   EMPSS02 and PUNCH EMPSS02;
    USER IS LSB                         turns on security for all other
      REGISTERED FOR ALL                verbs issued against EMPSS02
    PUBLIC ACCESS IS ALLOWED
      FOR DISPLAY.
```

Authorized Users

An authorized user for a specific schema or subschema is one whose association with the schema or subschema includes the verb used in the SCHEMA or SUBSCHEMA statement being processed. This authority is assigned through the REGISTERED FOR subclause of the USER clause in a previously-issued SCHEMA or SUBSCHEMA statement, as shown in the following examples:

```
ADD SUBSCHEMA NAME IS EMPSS01           assigns authority to KCO to
    USER NAME IS KCO                    use all verbs against EMPSS01
      REGISTERED FOR ALL.

ADD SUBSCHEMA NAME IS EMPSS02           assigns authority to WXE to
    USER NAME IS WXE                    access EMPSS02 with only those
      REGISTERED FOR PUBLIC ACCESS.     verbs specified in EMPSS02's
                                        PUBLIC ACCESS clause

ADD SCHEMA NAME IS EMPSCHM              assigns authority to ILI to
    USER NAME IS ILI                    DISPLAY and PUNCH EMPSCHM
      REGISTERED FOR DISPLAY.
```

Note: For more information about PUBLIC ACCESS and USER clauses, see "SCHEMA statement" in "SCHEMA statement" in Chapter 14, “Schema Statements".

Establishing Schema and Subschema Currency

You establish schema or subschema currency when you enter a SCHEMA or SUBSCHEMA statement. Once a specific schema or subschema becomes current, subsequent statements are applied to that schema or subschema.

There are two types of currency: update and display.

Type of Currency	Set by…	Allows…
Update	ADD SCHEMA/SUBSCHEMA or MODIFY SCHEMA/SUBSCHEMA	All operations against components
Display	Any schema or subschema statement (except DELETE)	Schema or subschema components to be displayed and punched

Example of Changes in Currency

The following example shows schema currency changes. Note that DISPLAY does not cancel update currency when the displayed schema was previously current for update.

EMPSCHM is current for display only; schema components cannot be modified.

```
dis schema empschm.
  dis area emp-demo-region.
  dis rec employee.
```

EMPSCHM is current for update and display; schema components can be added, modified, deleted, displayed, and punched.

```
mod schema empschm.
  del set ooak-skill.
  del set ooak-job.
  dis record job.
```

DEMOSCHM is current for both update and display; EMPSCHM has lost all currency.

```
mod schema demoschm.
  del set order-oremark.
  dis rec oremark.
```

DEMOSCHM remains current for both update and display; DISPLAY does not cancel update currency (for the same schema).

```
dis schema demoschm.
  del set product-item.
  del rec product.
  dis rec item.
```

EMPSCHM is current for display only; DEMOSCHM loses all currency; no schema is current for update.

```
dis schema empschm.
  dis area org-demo-region.
  dis set dept-employee.
  dis rec dept.
```

Reporting on Schema and Subschema Definitions

There are two methods of obtaining a report on a schema or subschema:

- Running the IDMSRPTS utility program
- Running the schema or subschema compiler in batch mode to produce an activity listing

More Information

- For more information about IDMSRPTS, see the *CA IDMS Utilities Guide*.
- For more information about schema and subschema compiler activity listings, see 10.7.2, “Schema and Subschema Listings".
- For more information about batch compiling, see Appendix E, “Batch Compiler Execution JCL".

Chapter 10: Using the Schema and Subschema Compilers

This section contains the following topics:

Overview

This chapter describes how to use the schema and subschema compilers, specifically, how to:

- Submit statements to the schema and subschema compilers
- Compile in batch and online environments
- Store a subschema load module
- Get a listing of a schema or subschema definition

Other information about the compiling environment is provided where appropriate.

More Information

- For descriptions of what the schema and subschema compilers do, see Chapter 10, “Using the Schema and Subschema Compilers".
- For the rules concerning the writing of user exits for the schema and subschema compiler, see Appendix G, "User-Exit Program for Schema and Subschema Compiler".

Online Compiling

You can use an online session to input source DDL statements that create, modify, delete, or display schema and subschema definitions.

Note: For more information about the batch alternative, see 10.3, "Batch Compiling".

An online session begins when the user signs on to the compiler, continues through any number of DDL operations, and ends when the user signs off from or otherwise terminates the compiler.

Starting a Session

To start an online session, do the following:

1. Sign on to the host TP monitor according to site-standard conventions.
2. Enter the task code for the compiler according to site-standard conventions. Task codes are SCHEMA for the schema compiler, SSC for the subschema compiler. A line identifying the compiler appears at the top of the screen.
3. Optionally, enter the SIGNON statement in the input/output area of the screen.
4. Optionally, enter the SET OPTIONS statement after the SIGNON statement to establish processing options for this session.

Note: For more information about SIGNON, SET OPTIONS, and other compiler-directive statements, see Chapter 11, “Compiler-Directive Statements”.

Submitting Statements

After you are signed on, you can enter ADD, MODIFY, DELETE, DISPLAY, and PUNCH statements (see Chapter 12, “Operations on Entities”).

Note: For more information about schema statements, see Chapter 14, “Schema Statements”. For more information about subschema statements, see Chapter 15, “Subschema Statements”.

Ending a Session

To end an online session, do the following:

1. Enter SIGNOFF in the input/output area, then press [Enter]. This erases the work file, terminates the full-screen editor, erases the session options, and displays a transaction summary.
2. Press [Clear]. This returns control to the system.

Note: To end a session and return control to the system without receiving a transaction summary, enter the END command on the top line of the screen instead of using SIGNOFF.

Recovering a Session

If the Compiler Abends

If the schema or subschema compiler terminates abnormally and you want to resume at the point before which you entered the last statement, enter the compiler's task code.

All updates made to the dictionary remain intact. Text changes made to the last screen are applied to your work file.

If the DC/UCF System Abends

If your system terminates abnormally, the work file and all session options are lost. Enter the compiler's task code to begin a new session.

Batch Compiling

You can use a batch stream to input source DDL statements that create, modify, delete, or display schema and subschema definitions.

Note: For more information about the online alternative, see 10.2, “Online Compiling”.

The following are the batch programs you use to compile source DDL statements for non-SQL databases:

- IDMSCHEM (batch program for schema compiling)
- IDMSUBSC (batch program for subschema compiling)

Running either of these programs in batch mode produces an activity listing (see 10.7.2, “Schema and Subschema Listings").

Note: For the JCL you need to run these compile programs under the central version or in local mode, see Batch Compiler Execution JCL.

Coding DDL Schema and Subschema Statements

This section describes how to submit logical DDL statements to the schema and subschema compilers. It describes common components of the DDL syntax, statement delimiters, symbols recognized as comments by the compilers, and input format.

Statement Components

Five Components

Most DDL statements consist of five components, in the following order (exceptions are presented later):

1. *Verb* (required) designates the specific operation to be performed by the statement: ADD, MODIFY, REPLACE, DELETE, DISPLAY, or PUNCH. Acceptable verb synonyms are shown in the following table.

Verb	Synonym
ADD	CREATE
MODIFY	ALTER
DELETE	DROP

2. *Entity type* (required) identifies the type of data in the dictionary that the selected operation will affect: SCHEMA, AREA, RECORD, SET, SUBSCHEMA, LOGICAL RECORD, or PATH-GROUP.

3. *Entity occurrence name* (required) identifies a specific instance of the named entity type.

4. *Optional clauses* provide qualifying data for each component occurrence. Optional clauses can be specified in any order, unless individual clause explanations state otherwise.

5. *Period* (required) signifies the end of the statement. The period can immediately follow the last word in the statement, can be separated from the last word by blanks, or can appear on a separate line.

   If you specify the SEMICOLON ALTERNATE clause of the SET OPTIONS compiler-directive statement, both the period (.) and the semicolon (;) will be recognized as statement terminators.

Example Statement

The following example illustrates the parts of the typical DDL statement:

```
ADD    SCHEMA      EMPSCHM    MEMO DATE IS 04/30/92       .
 ▲       ▲            ▲       ▲                   ▲        ▲
 |       |            |       └───────────────────┘        |
verb   entity       entity      optional clause       terminating
        type      occurrence                             period
                     name
```

Statement Exceptions

Exceptions to the syntax format rule stated above are clearly indicated in both the syntax layouts and the syntax explanations of the individual statements. Exceptions include the following:

- DELETE operations, which *must not* contain optional clauses (other than those needed to uniquely qualify the entity, such as VERSION, or satisfy security requirements, that is, PREPARED BY)
- VALIDATE, GENERATE, and REGENERATE, which do not name entities
- Carriage control statements (for compiler listings)
- Compiler-directive statements

Delimiting Statements

Required Delimiters

One or more blanks must be used as delimiters between words and clauses.

Optional Delimiters

Commas (,) and colons (:), are treated as blanks by the compilers and can be used as delimiters between words and clauses to enhance readability. You can also use a semicolon (;) as a delimiter if the SET OPTIONS statement does not set the SEMICOLON ALTERNATE END OF SENTENCE to ON.

End of Statement Delimiter

A period (.) signifies the end of the statement. You can also designate a semicolon (;) as an alternative statement terminator by specifying ON in the SEMICOLON ALTERNATE clause of the SET OPTIONS statement.

Compiler Comments

You can use the following symbols to begin a comment:

Symbol	Column
*+ (asterisk, plus)	Any
-- (hyphen, hyphen)	Any
* (asterisk)	1

CA IDMS/DB treats all remaining text on the input line as a comment.

Significance of *

An asterisk *as the first nonblank character of the input line* identifies the line as a compiler comment: lines beginning with an asterisk are ignored by the compiler.

Significance of *+

The combination of the asterisk and the plus sign *in columns one and two of an input line* identifies the line both as a comment line (because of the asterisk) and as a line not to be redisplayed.

Note: For more information about the compiler's ability to redisplay input, see the ECHO and LIST options in 11.5, “SET OPTIONS Statement".

Comment Lines in Messages and DISPLAY Output

The DDL compilers generate lines beginning with the *+ combination, as follows:

- **Messages**—All informational, warning, and error messages displayed by the compilers are preceded by *+.
- **DISPLAY output**—All output lines generated by a DISPLAY AS COMMENTS statement are preceded by *+. For DISPLAY AS SYNTAX, lines which contain information not directly associated with syntax statements are preceded by *+.

Example

In this example, the schema compiler ignores the WITHIN AREA clause when it processes the ADD RECORD statement:

```
ADD RECORD NAME IS EMPLOYEE
  LOCATION MODE IS CALC
    USING (ID-0415)
    DUPLICATES ARE NOT ALLOWED
  *+ WITHIN AREA EMP-DEMO-REGION .
```

Input Format

80-character Input

You can code statements in columns 1 through 80, or you can limit the input range using the INPUT COLUMNS clause of the SET OPTIONS statement. The maximum range is 1 through 80. The minimum number allowed between low and high columns is 10. The default depends on the mode in which the compiler is used:

- **Online default**—1 through 79 in 3270 full-screen mode; 1 through 80 for line devices
- **Batch default**—1 through 72

Multiline Input

DDL statements can be coded as multiple-line input. The four required statement components (verb, entity type, entity occurrence, and period) and most optional clauses can be continued from one line to the next, as long as words are not split (including user-supplied names in quotes). No continuation character is required.

Three examples of acceptable subschema DDL input are shown as follows:

- **Single line input:**

  ```
  add subschema name is empss01 schema is empschm.
  ```

- **Multiline input:**

  ```
  add subschema name is empss01
      schema is empschm.
  ```

- **Multiple statements per line:**

  ```
  signon user is msk. dis schema demoschm with none.
  dis area demoxarea. dis rec employee. dis set dept-employee.
  ```

Error Handling

An error is any condition that prevents the compiler from performing the requested operation. The user errors detected by the schema and subschema compilers fall into two major categories:

- Syntax errors
- Logic errors

The compilers check for both kinds of errors at the same time.

Syntax Errors

Syntax errors are those caused when you violate a clause's format rules (such as a misspelled keyword).

When you submit a statement as input, the compiler examines the statement, word by word, expecting specific combinations of keywords and expressions. This process is known as *parsing* the syntax.

The compiler expects a sentence to begin with a verb and end with a period. Words that can follow the verb vary, depending on the verb; words that can follow a component name vary, depending on the component type; and so on. If the compiler parses a clause or subclause successfully (that is, if keywords and expressions fall in the expected order), the compiler attempts to apply that clause. If not, the compiler processes the error.

Logic Errors

Logic errors are those caused when a syntactically correct clause requests an operation that is not practicable (such as a request to modify a nonexistent component).

When the DDL compiler attempts to satisfy a specific request, it may find that the request is not logical. When trying to determine the cause of a logic error, you should consider the following possibilities:

- Sentence—Logical errors can be caused by illogic of the sentence itself (for example, an attempt to modify a nonexistent component)
- Clause—Because the compiler examines each clause individually, logic errors can occur in individual clauses.
- Combination of sentence and clause—Logic errors can be caused by an illogical combination of otherwise correct clauses or statements, such as a SUBSCHEMA statement whose usage is DML followed by a LOGICAL RECORD statement.

Some logic errors are not detected when the statement is processed. Check for interdependence of components occurs when the VALIDATE statement is executed.

Example of a Logic Error

The following example shows a logic error. The first statement contains no errors; the second statement attempts to assign a record ID already assigned to a record. The compiler actions caused by the partially correct statement are shown at right.

```
add record department
  record id is 410
  location mode is calc using dept-id
    duplicates not allowed
  02 dept-id    pic x(4).
add record employee   ◄───────────────────── Puts record in dictionary
  record id is 410    ◄───────────────────── Produces error messages
  location mode is calc using emp-id  ◄───── Assigns location mode and
     duplicates not allowed                     duplicates option to record
  02 emp-id    pic 9(4).  ◄───────────────── Associates elements with
                                                the record
```

After the processing is complete, the dictionary contains a partial description of the EMPLOYEE record. To complete the description, you should issue the following statement:

```
modify record employee record id is 411.
```

You can specify any record ID other than 410. Because the location mode and elements already are part of the record description in the dictionary, you would not need to recode them.

FORWARD SPACING Message

The message FORWARD SPACING TO NEXT PERIOD indicates that the compiler cannot continue processing the statement. For example, if the compiler detects an invalid password, the compiler must reject all clauses in the statement. To resume processing, the compiler searches for the end of the statement (the period) and begins with the next keyword. This message is issued when the compiler is checking either identification or security at the *beginning* of the statement. Consequently, no partial updates occur when this message is issued.

More Information about Messages

For detailed information about error/status messages, see the *CA IDMS Messages and Codes Guide* document.

Coding Keywords, Variables, and Comment Text

A DDL input statement contains keywords and variables. This section provides rules for:

- Coding keywords
- Forming entity-occurrence names
- Using quotes in user-supplied names
- Coding comments in schema and subschema descriptions

Coding Keywords

Keywords are predefined names or special characters that appear in syntax diagrams. Required letters appear in uppercase. Optional letters are in lowercase.

Abbreviations

Keywords can be spelled in full, or they can be abbreviated to a minimum of three characters if no other word in the same syntactical position can be abbreviated identically. The keywords ELEMENT and VERSION are exceptions to the three-character minimum requirement and can be abbreviated to EL and V, respectively.

Coding Entity-Occurrence Names

An entity-occurrence name is the name you provide to a schema or subschema entity, such as the schema itself, a schema area, and a schema record.

Valid Characters

The following are valid characters to include in entity-occurrence names:

- Letters (A through Z)

  Lowercase letters in entity names are translated to uppercase.
- Digits (0 through 9)
- At sign (@)
- Dollar sign ($)
- Pound sign (#)
- Hyphen (-)

The first character of an identifier must be a letter, @, $, or #. A hyphen cannot be the last character and cannot follow another hyphen.

Note: Element name can also begin with a digit (0 through 9).

Program Language Restrictions

Because the DDL compilers cannot anticipate which programming languages will use which records and elements, the user is responsible for ensuring that record and element names follow the character set and word length rules and do not duplicate any of the reserved words of the specific compiler or assembler.

Maximum Length

The maximum length of an entity-occurrence name depends on the entity. Syntax rules for each entity indicate length restrictions.

Avoid Using Keywords

Keywords recognized by a DML processor *may* inhibit the processor's operation when used as entity-occurrence names. Keywords will pass successfully through the processor under some conditions, but not under others. Consequently, avoid using keywords as entity-occurrence names.

Coding User-Supplied Values

A user-supplied value is any text value, except an entity-occurrence name, that you supply in a DDL statement. For example, your user ID and password are user-supplied values. So are character-string literals used in Boolean expressions and descriptive text for schema and subschema entities.

Lowercase letters are retained in user-supplied values which are enclosed in quotes (such as comments). In values not enclosed in quotes, lowercase letters are translated to uppercase.

Using Quotes for Special Characters

The coding rules listed for entity-occurrence names apply to user-supplied values. In addition, you can use quotation marks in order to use special characters. The DDL compilers treat these characters as *special characters* :

- Comma (,)
- Period (.)
- Semicolon (;)
- Apostrophe (')
- Parenthesis ((and))
- Colon (:)
- The quote character (' or ")

Default Quotation Mark

The single quotation mark (') is the default quote character established during installation. You can specify the double quotation mark (") as the quote character by means of the SET OPTIONS statement.

Embedding the Quote Character

When the quote character is to be embedded in a user-supplied name, it must appear twice for each occurrence in the original name. For example, the name MARY'S PROGRAM should be input as 'MARY''S PROGRAM' if the single quotation mark (') has been designated as the quote character, and as "MARY'S PROGRAM" if the double quotation mark (") has been designated as the quote character.

Code the Closing Quote

If the closing quote is omitted from a quoted literal, the literal is interpreted as including everything to the end of the input column range.

Nullifying Existing Values

Two quote characters with no space between them is a *null string*. Null strings can be used for nulling out existing values. Note that the null string does not null lines of comment text in COMMENTS clauses; it creates one blank line.

Coding Comment Text

You can add comments to SCHEMA, SUBSCHEMA, RECORD element substatement, and LOGICAL record definitions in the COMMENTS clause. Rules for coding *comment-text* appear next.

Text Can Be Any Length

Text can extend to any length. Code as many lines as are necessary to document the entity.

Use Quotes on Each Line

A quote must precede the text of each line; ending quotes on each line are optional. When COMMENTS is the last clause in the statement and the *terminating* quote (at the end of the *last line*) is omitted, code the period on a separate line; otherwise, the compiler treats the period as part of the comment.

Multiline Input

When text extends beyond the first line of input, each subsequent line must begin with a character indicating either continuation or concatenation, as follows:

Line type	Symbol	Meaning
Continuation	Hyphen (-)	Compilers treat the new line as a continuation of comment text
Concatenation	Plus (+)	Compilers append the new line to the preceding line of comment text; any number of text lines can be concatenated, provided that their combined length does not exceed 80 bytes

Examples

The following example of the SUBSCHEMA statement compares valid omissions of a terminating quote with an invalid omission. In the third statement, the subschema compiler assumes that the period is part of the comment and that ADD RECORD was meant to be a clause of the SUBSCHEMA statement; because this is not valid syntax, the compiler flags ADD RECORD as an error.

- **Valid**:

```
modify subschema name is empss01
   usage is dml
   comments 'This subschema is used only in emergencies
   .

modify subschema name is empss02
   comments 'This subschema will be obsolete by August.
         usage is dml.
```

- **Invalid**:

```
modify subschema name is empss03
   comments 'This subschema will be obsolete by August.
add record name is employee.
```

The following example illustrates continuation and concatenation:

- **As input**:

```
add subschema name is empss01
  comments 'Includes the entire '
  + 'employee '
  + 'database.'
  - 'Ron and Jan '
  + 'are responsible for this subschema.'.
```

- **As output of DISPLAY or PUNCH**:

```
ADD SUBSCHEMA NAME IS EMPSS01
    COMMENTS 'INCLUDES THE ENTIRE EMPLOYEE DATABASE.'
    - 'RON AND JAN ARE RESPONSIBLE FOR THIS SUBSCHEMA.'.
```

Compiler-Directive Statements

Using DDL compiler-directive statements, you can sign on to and sign off the compiler and set and view compiler defaults.

- SIGNON identifies the environment in which the compiler is to execute
- SIGNOFF terminates the compiler
- SET OPTIONS establishes defaults for execution of the compiler
- DISPLAY/PUNCH OPTIONS informs you of the defaults currently in effect
- INCLUDE instructs the compiler to use as input the code found in a dictionary source module

You can issue these statements, including SIGNON and SIGNOFF, either online or in batch mode.

Note: For more information about and syntax for compiler directives, see Chapter 11, “Compiler-Directive Statements".

Output From the Compilers

This section describes the source code, load modules, and hardcopy listings created by the schema and subschema compilers.

Source Code and Load Modules

Schema Definition

The schema compiler creates and maintains schemas in the DDLDML area of the dictionary in source form only; no schema load module ever exists. The schema is not used at runtime.

The dictionary can contain any number of schemas to define different versions of one database or to define several independent databases.

A schema is identified by a name and version number, the combination of which must be unique.

Subschema Definition

The subschema compiler creates subschemas in source *and* load module forms in the dictionary. Source is stored in the DDLDML area. Load modules are stored in the DDLDCLOD area.

Storing a Subschema Load Module in a Load Library

Optionally, you can store a subschema load module in a load library. To do this, perform the following steps:

1. *Punch the load module using the subschema or DDDL compiler*—Submit the PUNCH LOAD MODULE statement to obtain an object deck of the subschema.
2. *Link edit the object deck into the load library*—Execute the operating system's linkage editor, using as input the object deck produced by the PUNCH LOAD MODULE statement.

Load Modules at Runtime

At runtime, the subschema load module can reside in either the dictionary load area or a load library. If it resides in both places, CA IDMS/DB uses the first one it finds based on the loadlist and dictionary established for your session.

If you are using CA OLQ, you must keep the subschemas being accessed by CA OLQ in the dictionary load area.

Note: For more information about loading, see the *CA IDMS System Generation Guide* document.

Schema and Subschema Listings

Running the schema and subschema *batch* compile programs produces hardcopy listings, as follows:

- IDMSCHEM produces the Schema Compiler Activity List
- IDMSUBSC produces the Subschema Compiler Activity List

Contents of a Listing

The Schema Compiler Activity List and the Subschema Compiler Activity List each contain the following information:

- Heading—The top of each page of the listing contains the name of the software component (IDMSCHEM or IDMSUBSC), release number, name of the listing, date, time, and page number.
- Input listing—The body of the printout contains a line for each line of source code you entered. Column 1 shows the compiler-assigned line number. Column 10 shows the text of schema source code.
- Warning and error messages—These messages are interspersed in the body of the report, as needed. See the *CA IDMS Messages and Codes Guide* for descriptions of the compiler messages.
- Transaction summary—The transaction summary indicates the number of schemas/subschemas compiled and the number of error and warning messages issued

Format-Control Statements for Listings

You can use the SKIP and EJECT statements to format the schema and subschema listings.

SKIP

Use the SKIP1/2/3 statement to insert 1, 2, or 3 blank lines after the line on which you entered the statement.

Do not use a terminating period with SKIP and leave no space between SKIP and the number you specify. For example, code SKIP2, *not* SKIP 2.

EJECT

Use the EJECT statement to force a new page. Printing on the new page begins with the statement following the EJECT statement.

Do not use a terminating period with EJECT.

Chapter 11: Compiler-Directive Statements

This section contains the following topics:

Overview

Compiler-directive statements are any statements you issue to the schema or subschema compiler that do not produce schema or subschema definitions. The following table describes the compiler-directive statements presented in this chapter:

Purpose	Statement	Description
Signon	SIGNON	Identifies the user and the environment in which the compiler is to execute.
Signoff	SIGNOFF	Terminates the compiler.
Set compiler default values	SET OPTIONS	Establishes defaults for execution of the compiler.
Include module source	INCLUDE	Instructs the compiler to use as input the code found in a dictionary source module.
Display all entity occurrences	DISPLAY/PUNCH ALL	Displays or punches all occurrences of a schema or subschema entity.
Display IDD definitions	DISPLAY/PUNCH IDD	Displays or punches the definition of an entity occurrence.

The DBA can issue these statements, including SIGNON and SIGNOFF, either online or in batch mode. Compiler-directive statements are described in alphabetical order.

DISPLAY/PUNCH ALL Statement

The DISPLAY/PUNCH ALL statement displays all occurrences of an entity related to the schema or subschema compiler from which the statement is issued.

Syntax

```
►►─┬─ DISplay ─┬──┬─ ALL ──────────────────────────┬─ entity-type ───────►
   └─ PUNch ───┘  └─┬─ FIRst ───────────────────┬─┘
                    ├─ LASt ──┤    ┌─ 1 ◄────────┤
                    ├─ NEXt ──┤    └─ entity-count ┘
                    └─ PRIor ─┘

 ►─┬────────────────────────────────────────────────────────┬─────────►
   └─ PREpared by user-id ─┬─────────────────────────────┬─┘
                           └─ PASsword is password ───────┘

 ►─┬─────────────────────────────────┬────────────────────────────────►
   └─ WHEre conditional-expression ──┘

 ►─┬──────────────────────┬───────────────────────────────────────────►
   └─ VERB ─┬─ ADD ───────┤
            ├─ MODify ────┤
            ├─ REPlace ───┤
            ├─ DELete ────┤
            ├─ DISplay ───┤
            └─ PUNch ─────┘

 ►─┬────────────────────────┬─────────────────────────────────────────►
   └─ AS ─┬─ COMments ──────┤
          └─ SYNtax ────────┘

 ►─┬──────────────────────────────────────┬──────────────────────────►◄
   └─ TO ─┬─ module-specification ────────┤
          └─ SYSpch ──────────────────────┘
```

Expansion of *conditional-expression*

```
►►─┬─ mask-comparison ─────────────────────────────────────┬──────────►
   ├─ value-comparison ────────────────────────────────────┤
   └─┬───────┬─ ( ─┬─ mask-comparison ──┬─ ) ──────────────┘
     └─ NOT ─┘     └─ value-comparison ─┘

 ►─┬──────────────────────────────────────────────────────────────┬──►◄
   │ ┌────────────────────────────────────────────────────────────┤ │
   └─▼─┬─ AND ─┬─┬─ mask-comparison ────────────────────────────┬─┴─┘
       └─ OR ──┘ ├─ value-comparison ───────────────────────────┤
                 └─┬───────┬─ ( ─┬─ mask-comparison ──┬─ ) ─────┘
                   └─ NOT ─┘     └─ value-comparison ─┘
```

Expansion of *mask-comparison*

```
►►─── entity-option-keyword ──────────────────────────────────────────►

 ►─┬─ CONTAINs ─┬─ 'mask-value' ─────────────────────────────────────►◄
   └─ MATCHES ──┘
```

Expansion of *value-comparison*

```
►►─┬─ 'character-string-literal' ─┬──────────────────────────────────────────►
   ├─ numeric-literal ────────────┤
   └─ entity-option-keyword ──────┘

►─┬─ IS ─┬───────┬──────┬─┬─ 'character-string-literal' ─┬───────────────►◄
  │      └─ NOT ─┘      │ ├─ numeric-literal ────────────┤
  ├─ NE ────────────────┤ └─ entity-option-keyword ──────┘
  └─┬───────┬─┬─┬─ EQ ─┬─┬┘
    └─ NOT ─┘ │ └─ = ──┘ │
              ├─┬─ GT ─┬─┤
              │ └─ > ──┘ │
              ├─┬─ LT ─┬─┤
              │ └─ < ──┘ │
              ├─── GE ───┤
              └─── LE ───┘
```

Parameters

ALL

Lists all occurrences of the requested entity type that the current user is authorized to display.

Online users: With a large number of entity occurrences, ALL may slow response time.

FIRst

Lists the first occurrence of the named entity type.

LASt

Lists the last occurrence of the named entity type.

NEXt

Lists the next occurrence of the named entity type.

PRIor

Lists the prior occurrence of the named entity type.

entity-count

Specifies the number of occurrences of the named entity type to list. 1 is the default.

entity-type

Identifies the entity type or entity synonym that is the object of the DISPLAY/PUNCH ALL request. Valid values for each compiler appear in the table under "Usage" in this section.

WHEre *conditional-expression*

Specifies criteria to be used by the compiler in selecting occurrences of the requested entity type.

The outcome of a test for the condition determines which occurrences of the named entity type the schema or subschema compiler selects for display.

mask-comparison

Compares an entity type operand with a mask value.

entity-option-keyword

Identifies the left operand as a syntax option associated with the named entity type. The table in the "Usage" section lists valid options for each entity type.

CONTAINs

Searches the left operand for an occurrence of the right operand. The length of the right operand must be less than or equal to the length of the left operand. If the right operand is not contained entirely in the left operand, the outcome of the condition is false.

MATCHES

Compares the left operand with the right operand one character at a time, beginning with the leftmost character in each operand. When a character in the left operand does not match a character in the right operand, the outcome of the condition is false.

'*mask-value*'

Identifies the right operand as a character string; the specified value must be enclosed in quotation marks. *Mask-value* can contain the following special characters:

Special Characters	Description
@	Matches any alphabetic character in *entity-option-keyword*.
#	Matches any numeric character in *entity-option-keyword*.
*	Matches any character in *entity-option-keyword*.

value-comparison

Compares values contained in the left and right operands based on the specified comparison operator.

'*character-string-literal*'

Identifies a character string enclosed in quotes.

numeric-literal

Identifies a numeric value.

entity-option-keyword

Identifies a syntax option associated with the named entity type; valid options for each entity type are listed in the table presented in the "Usage" section.

IS

Specifies that the left operand must equal the right operand for the condition to be true.

NE

Specifies that the left operand must *not* equal the right operand for the condition to be true.

EQ/=

Specifies that the left operand must equal the right operand for the condition to be true.

GT/>

Specifies that the left operand must be greater than the right operand for the condition to be true.

LT/<

Specifies that the left operand must be less than the right operand for the condition to be true.

GE

Specifies that the left operand must be greater than or equal to the right operand for the condition to be true.

LE

Specifies that the left operand must be less than or equal to the right operand for the condition to be true.

NOT

Specifies that the opposite of the condition fulfills the test requirements. If NOT is specified, the condition must be enclosed in parentheses.

AND

Indicates the expression is true only if the outcome of both test conditions is true.

OR

Indicates the expression is true if the outcome of either one or both test conditions is true.

Note: For descriptions of the remaining DISPLAY parameters, see 12.5, “DISPLAY/PUNCH Operations".

Usage

Limiting the Number of Records Read

You can limit the number of entity occurrences CA IDMS/DB reads for a DISPLAY ALL request by setting two options in the SET OPTIONS statement:

- DISPLAY ALL LIMIT IS ON activates interrupt processing.
- INTERRUPT COUNT IS *interrupt-count* terminates the DISPLAY ALL request when the number of occurrences read exceeds the interrupt limit, whether or not the occurrences meet the criteria of an associated WHERE clause. If you set the interrupt count to NULL, CA IDMS/DB will reject DISPLAY ALL requests.

Type of Display Depends on Compiler and Entity

The compilers display the entity occurrences as syntax or as comments depending on the entity type requested and the compiler in which the DISPLAY/PUNCH ALL statement is issued, as shown in the following table

Note: C means display as comments; S means display as syntax, if requested.

Entity type	Compiler Schema	Compiler Subschema
ATTRIBUTES	C	C
CLASSES	C	C
ELEMENTS	C	
ELEMENT SYNONYMS	C	
LOAD MODULES		S
RECORDS	C	

Entity type	Compiler Schema	Compiler Subschema
RECORD SYNONYMS	C	
SCHEMAS	S	C
SUBSCHEMAS	C	S
USERS	C	C

Output Contains Only Enough Information to Display/Punch Entity

Output produced by DISPLAY or PUNCH ALL consists only of the information necessary to execute a DISPLAY/PUNCH request for each entity occurrence. For example, RECORD occurrences are displayed with their name and version, and ATTRIBUTE occurrences with their name and class. In an online session, the user can execute the displayed statements by pressing [Enter]. This two-step process allows the user to scan the names of entity occurrences related to the compiler in which the statement is issued.

Valid Entity Option Keywords for Conditional Expressions

The following table lists entity type options that you can specify in a conditional expression.

Entity type	Option	Entity type	Option
ATTribute User-defined entity	Entity-type name PREpared by REVised by DATe last UPDated MONth last UPDated DAY last UPDated YEAr last UPDated DATe CREated MONth CREated DAY CREated YEAr CREated CLAss name	CLAss	Entity-type name PREpared by REVised by DATe last UPDated MONth last UPDated DAY last UPDated YEAr last UPDated DATe CREated MONth CREated DAY CREated YEAr CREated

Entity type	Option	Entity type	Option
ELement RECord USEr	Entity-type name Version PREpared by REVised by DATe last UPDated DATe CREated DEScription FULl name (users only)	ELement SYNonym	ELement SYNonyn name ELement NAMe Version PREpared by REVised by DATe last UPDated MONth last UPDated DAY last UPDated YEAr last UPDated DATe CREated MONth CREated DAY CREated YEAr CREated DEScription
RECord SYNonym	SYNonym NAMe RECord NAMe Version PREfix SUFfix VIEw id PREpared by REVised by DATe last UPDated MONth last UPDated DAY last UPDated YEAr last UPDated DATe CREated MONth CREated DAY CREated YEAr CREated DEScription	SUBschema	Entity-type name PREpared by REVised by DATe last UPDated MONth last UPDated DAY last UPDated YEAr last UPDated DATe CREated MONth CREated DAY CREated YEAr CREated DEScription SCHema NAMe SCHema Version

Entity type	Option	Entity type	Option
SCHema	Entity-type name	LOAd module	Entity-type name
	PREpared by		Version
	REVised by		DATe COMpiled
	DATe last UPDated		MONth COMpiled
	MONth last UPDated		DAY COMpiled
	DAY last UPDated		YEAr COMpiled
	YEAr last UPDated		
	DATe CREated		
	MONth CREated		
	DAY CREated		
	YEAr CREated		
	DATe COMpiled		
	MONth COMpiled		
	DAY COMpiled		
	YEAr COMpiled		
	DEScription		

Default Order of Precedence Applied to Logical Operators

Conditional expressions can contain a single condition, or two or more conditions combined with the logical operators AND or OR. The logical operator NOT specifies the opposite of the condition. The compiler evaluates operators in a conditional expression 1 at a time, from left to right, in order of precedence. The default order of precedence is as follows:

- MATCHES or CONTAINS keywords
- EQ, NE, GT, LT, GE, LE operators
- NOT
- AND
- OR

If parentheses are used to override the default order of precedence, the compiler evaluates the expression within the innermost parentheses first.

Example

The following example displays all records prepared by user JKD since June 1, 1986:

```
display all records
  where prepared by eq 'jkd' and
  year created ge '86' and
  month created ge '06' as syntax.
```

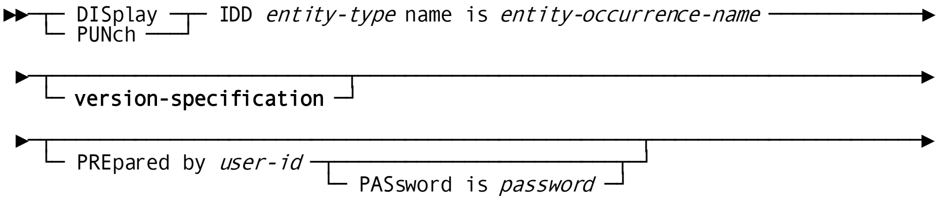
DISPLAY/PUNCH IDD Statement

The DISPLAY/PUNCH IDD statement displays the dictionary definition of an entity occurrence related to the schema or subschema compiler. The output is displayed as comments.

The following table lists the entity definitions that the schema and subschema compilers display:

Entity Type	Schema Compiler	Subschema Compiler
ATTRIBUTE	X	X
CLASS	X	X
ELEMENT	X	
ELEMENT SYNONYM	X	
RECORD	X	
RECORD SYNONYM	X	
USER	X	X
LOAD MODULE		X

Syntax



```
►─┬───────────────────────────────────────────────────┬─────────────────────────►
  │                      ┌─────────────────────────┐  │
  └─┬─ WITh ─────────┬──▼─ entity-option-keyword ─┴──┘
    ├─ ALSo WITh ────┤
    └─ WITHOut ──────┘

►─┬──────────────────────────┬──────────────────────────────────────────────────►
  └─ VERB ─┬─ ADD ───────────┤
           ├─ MODify ──────┤
           ├─ REPlace ─────┤
           ├─ DELete ──────┤
           ├─ DISplay ─────┤
           └─ PUNch ───────┘

►─┬──────────────────────────────────────┬──────────────────────────────────────►◄
  └─ TO ─┬─ module-specification ─┬──┘
         └─ SYSpch ───────────────┘
```

Parameters

DISPLAY/PUNCH IDD

Lists or punches an IDD definition as comments.

entity-type

Specifies one of the entity types listed in the previous table.

entity-occurrence-name

Names an existing occurrence of the specified entity type.

Note: For descriptions of the remaining parameters, see 12.5, “DISPLAY/PUNCH Operations”.

Example

In the following example, the dictionary definition of version 100 of the DEPARTMENT record is requested from the schema compiler.

```
display idd record department version 100.
```

Note: For more information about DISPLAY/PUNCH syntax options, see Chapter 12, “Operations on Entities".

INCLUDE Statement

The INCLUDE statement temporarily suspends input to the schema or subschema compiler and retrieves, as input to the compiler, source statements from an existing source module in the dictionary.

Syntax

```
►►── INCLUDe module-specification ──────────────────────────────────────►◄
```

Parameters

INCLUDE *module-specification*

Includes in the current input file the source statements associated with the named module.

Note: Expanded syntax for *module-specification* is presented in Chapter 13, “Parameter Expansions".

Usage

Restrictions on Source Module Statements

The source module can contain any number of DDL statements; the following restrictions apply:

- INCLUDE statements cannot appear within the source module; that is, INCLUDE statements cannot be nested.
- The included module cannot update its own source. This restriction applies to the PUNCH statements of the DDL compilers, since they are capable of updating the module source.

  For example, the statement INCLUDE MODULE RECSRC. is unacceptable if the module RECSRC contains the source statement PUNCH RECORD EMPLOYEE TO MODULE RECSRC..

Compiler Continues Processing Statements Following INCLUDE

When all the module source has been processed, the compiler continues processing with the source statement immediately following the INCLUDE statement.

If the Source Module Contains a SIGNON Statement

If the module source being included contains a SIGNON statement to another dictionary, the DDL compiler terminates the INCLUDE operation, processes the SIGNON statement, and continues processing with the DDL statement immediately following the INCLUDE.

Example

Sample Session

The following example illustrates a schema compiler session in which the user requests the compiler to include source statements from the module EMPREC-SRC version 1:

- **IDD DDDL definition of module EMPREC-SRC**:

```
signon dict=empdict.
add module emprec-src version 1
   module source follows
      add record name is employee
          share structure of record employee
                of schema srcschm version 10.
      .
      .
      .
   msend.
```

- **Schema compiler DDL source**:

```
signon dict=empdict.
modify schema empschm version 7.
include module emprec-src version 1.
display record employee.
.
.
.
signoff.
```

Note: For more information about defining modules, see the *CA IDMS IDD DDDL Reference Guide*.

SET OPTIONS Statement

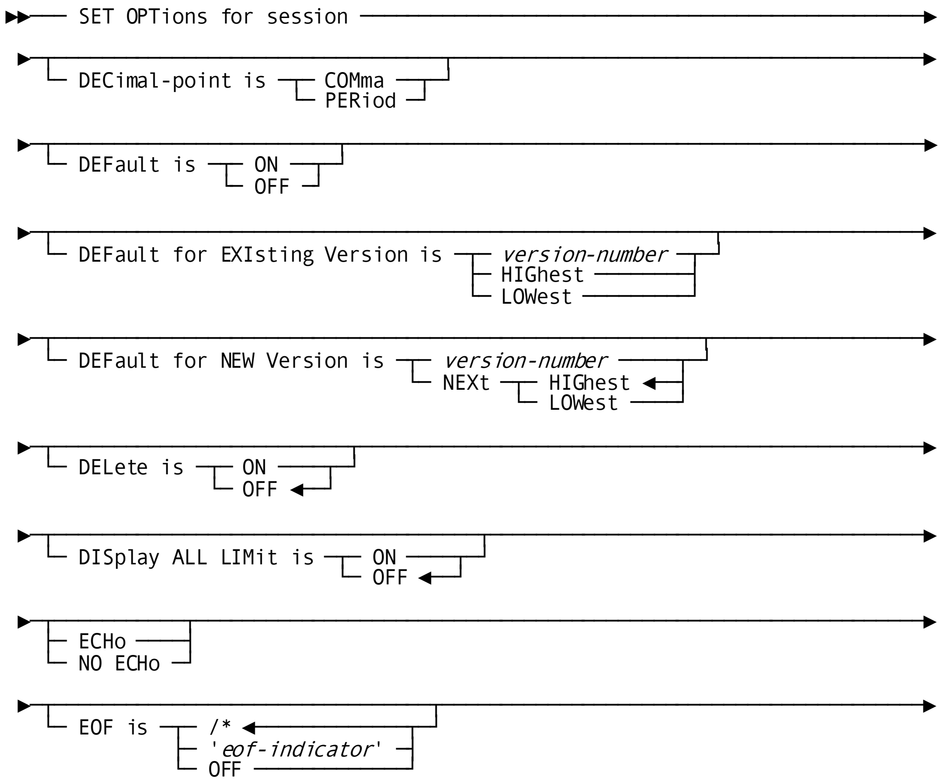
The SET OPTIONS statement allows a user to establish the following processing options for an individual session:

- Identification of the user who is adding, modifying, deleting, punching, or displaying component descriptions
- Quote character
- Decimal point character
- Characters for delimiting an input file
- Disposition of ADD statements issued for existing components

- Starting and incremental line numbers for record elements and for lines of comment text
- Compiler output format
- Conventions for specifying version numbers for schemas, records, and programs named in DDL statements
- Destination of punched descriptions
- Format of displayed or punched descriptions (syntax or comments)
- Information to be included in displayed or punched descriptions
- Automatic subschema load module deletion

Syntax

SET OPTIONS Statement



```
►►─── SET OPTions for session ──────────────────────────────────────────►

►─┬─────────────────────────────────────────────────────────────────────►
  └ DECimal-point is ─┬─ COMma ──┬┘
                      └─ PERiod ─┘

►─┬─────────────────────────────────────────────────────────────────────►
  └ DEFault is ─┬─ ON ──┬┘
                └─ OFF ─┘

►─┬─────────────────────────────────────────────────────────────────────►
  └ DEFault for EXIsting Version is ─┬─ version-number ─┬┘
                                     ├─ HIGhest ────────┤
                                     └─ LOWest ─────────┘

►─┬─────────────────────────────────────────────────────────────────────►
  └ DEFault for NEW Version is ─┬─ version-number ─────────┬┘
                                └─ NEXt ─┬─ HIGhest ◄─┬────┘
                                         └─ LOWest ───┘

►─┬─────────────────────────────────────────────────────────────────────►
  └ DELete is ─┬─ ON ────┬┘
               └─ OFF ◄──┘

►─┬─────────────────────────────────────────────────────────────────────►
  └ DISplay ALL LIMit is ─┬─ ON ────┬┘
                          └─ OFF ◄──┘

►─┬─────────────────────────────────────────────────────────────────────►
  ├ ECHo ────┤
  └ NO ECHo ─┘

►─┬─────────────────────────────────────────────────────────────────────►
  └ EOF is ─┬─ /* ◄─────────────────┬┘
            ├─ 'eof-indicator' ─────┤
            └─ OFF ─────────────────┘
```

```
►─┬───────────────┬────────────────────────────────────────────────────────►
  ├─ HEAder ──────┤
  └─ NO HEAder ───┘

►─┬───────────────────────────────────────────────────────────────────┬────►
  └─ INPut columns are start-column-number THRu end-column-number ────┘

►─┬──────────────────────────────────────────────────┬─────────────────────►
  └─ INTerrupt COUnt is ─┬─ interrupt-count ────┬────┘
                         └─ NULl ◄──────────────┘

►─┬─────────────────────────────────┬──────────────────────────────────────►
  └─ LINes per page is line-count ──┘

►─┬─────────────┬──────────────────────────────────────────────────────────►
  ├─ LISt ──────┤
  └─ NO LISt ───┘

►─┬─────────────────────────────────┬──────────────────────────────────────►
  └─ OUTput line size is ─┬─ 80 ──┬─┘
                          └─ 132 ─┘

►─┬──────────────────────────┬─────────────────────────────────────────────►
  └─ user-specification ─────┘

►─┬───────────────┬────────────────────────────────────────────────────────►
  ├─ PROmpt ──────┤
  └─ NO PROmpt ───┘

►─┬─────────────────────────────────────────────┬──────────────────────────►
  └─ PUNch TO ─┬─ module-specification ─────┬───┘
               └─ SYSpch ───────────────────┘

►─┬──────────────────────┬─────────────────────────────────────────────────►
  └─ QUOte is ─┬─ ' ─┬───┘
               └─ " ─┘

►─┬───────────────────────────────┬────────────────────────────────────────►
  └─ REGistration OVErride ───────┘

►─┬─────────────────────────────────────────────────────────┬──────────────►
  └─ SEMicolon alternate end of sentence is ─┬─ ON ────┬────┘
                                             └─ OFF ◄──┘

►─┬───────────────────────────────────────┬────────────────────────────────►
  └─ SEQuence is sequence-number ─────────┘

►─┬───────────────────────────────────────────────────────┬────────────────►
  └─ USEr signon OVErride is ─┬─┬─ ALLowed ◄───┬──┬───────┘
                              │ └─ ON ─────────┘  │
                              └─┬─ NOT ALLowed ─┬─┘
                                └─ OFF ─────────┘

►─┬────────────────────────────────┬───────────────────────────────────────►◄
  │  ┌──────────────────────────┐  │
  └─▼─ DISplay display-options ─┴──┘
```

DISPLAY/PUNCH OPTIONS Statement

```
►►─┬─ DISplay ─┬─ OPTions ─┬───────────────────────────────┬────────────►
   └─ PUNch ───┘           └─ for ─┬─ SESsion ◄───┬────────┘
                                   └─ DICtionary ─┘

 ►─┬────────────────────────────┬──────────────────────────────────────►
   └─┬─ WITh ─────┬─ DETails ───┘
     └─ WITHOut ──┘

 ►─┬─────────────────────────┬─────────────────────────────────────────►
   └─ AS ─┬─ COMments ─┬─────┘
          └─ SYNtax ───┘

 ►─┬─────────────────────────────────────┬─────────────────────────────►◄
   └─ TO ─┬─ module-specification ─┬─────┘
          └─ SYSpch ───────────────┘
```

Expansion for ***display-options***

```
►►─┬──────────────────────────────────────────────────────────────┬────►
   │                       ┌─────────────────────────────────┐    │
   └─┬─ WITh ◄──────┬─────▼─┬─ ALL COMment TYPes ─────────┬──┴────┘
     ├─ ALSo WITh ──┤       ├─ AREas ─────────────────────┤
     └─ WITHOut ────┘       ├─ ATTributes ────────────────┤
                            ├─ COMments ──────────────────┤
                            ├─ CULprit headers ───────────┤
                            ├─ DEFinitions ───────────────┤
                            ├─ DETails ───────────────────┤
                            ├─ ELements ──────────────────┤
                            ├─ HIStory ───────────────────┤
                            ├─ LRS ───────────────────────┤
                            ├─ OLQ headers ───────────────┤
                            ├─ PATh-groups ───────────────┤
                            ├─ PROgrams ──────────────────┤
                            ├─ RECords ───────────────────┤
                            ├─ SCHemas ───────────────────┤
                            ├─ SETs ──────────────────────┤
                            ├─ SHAred structures ─────────┤
                            ├─ SUBschemas ────────────────┤
                            ├─ SYMbols ───────────────────┤
                            ├─ SYNonyms ──────────────────┤
                            ├─┬─ USEr DEFINED COMments ─┬─┤
                            │ └─ UDCS ──────────────────┘ │
                            ├─ USErs ─────────────────────┤
                            ├─ ALL ◄──────────────────────┤
                            └─ NONE ──────────────────────┘

 ►─┬────────────────────────────┬──────────────────────────────────────►
   └─ VERB ─┬─ ADD ────────┬────┘
            ├─ MODify ─────┤
            ├─ DELete ─────┤
            ├─ DISplay ────┤
            └─ PUNch ──────┘

 ►─┬─────────────────────────┬─────────────────────────────────────────►◄
   └─ AS ─┬─ COMments ─┬─────┘
          └─ SYNtax ───┘
```

Parameters

SET OPTions for session

Establishes the defaults that govern a single session. All other executions of the compiler are unaffected by the options specified in this statement.

DECimal-point is COMma

Designates a comma as the character that represents a decimal point in DDL source statements.

When DECIMAL-POINT IS COMMA is in effect, a comma (,) is interpreted as a decimal point, and a period (.) is interpreted as an insertion character.

DECimal-point is PERiod

Designates a period as the character that represents a decimal point in DDL source statements.

When DECIMAL-POINT IS PERIOD is in effect, a period is interpreted as a decimal point, and a comma is interpreted as an insertion character.

DEFault is ON

Specifies that the compiler will accept ADD statements that identify established components and will interpret them as MODIFY statements. A warning message is issued when this occurs.

DEFault is OFF

Specifies that the compiler will *not* accept ADD statements that identify established components. The compiler issues an error message and terminates processing of the statement in error.

DEFault for EXIsting Version is

Establishes a default version number for existing schemas, records, programs, and source modules named in DDL statements. If a statement identifies an existing schema, record, or program without a version number, the compiler treats the statement as though it were coded with a VERSION clause in the format specified in the DEFAULT FOR EXISTING VERSION option. Version numbers must fall within the range 1 through 9999, whether specified explicitly or in relation to existing versions.

version-number

Specifies an explicit version number and must be an unsigned integer in the range 1 through 9999. If a subsequent DDL statement references an existing schema, record, or program without including a version number, the compiler selects the version number specified by *version-number*.

HIGhest

Specifies the highest existing version number for the named schema, record, or program. If a subsequent DDL statement references an existing schema, record, or program without including a version number, the compiler selects the highest existing version number for that schema, record, or program.

LOWest

Specifies the lowest existing version number for the named schema, record, or program. If a subsequent DDL statement references an existing schema, record, or program without including a version number, the compiler selects the lowest existing version number for that schema, record, or program.

DEFault for NEW Version is

Establishes a default version number for schemas being added to the dictionary. If an ADD SCHEMA statement names a schema without a version number, the compiler treats the statement as though it were coded with a VERSION clause in the format specified in the DEFAULT FOR NEW VERSION option. Version numbers must fall within the range 1 through 9999, whether specified explicitly or in relation to existing versions.

version-number

Specifies an explicit version number and must be an unsigned integer in the range 1 through 9999. If a subsequent ADD SCHEMA statement names a schema without including a version number, the compiler assigns the version number specified by *version-number*.

next HIGhest

Specifies the highest version number assigned to *schema-name* plus one. If a subsequent ADD SCHEMA statement names a schema without including a version number, the compiler assigns the highest existing version number for that schema name plus one.

next LOWest

Specifies the lowest version number assigned to *schema-name* minus one. If a subsequent ADD SCHEMA statement names a schema without including a version number, the compiler assigns the lowest existing version number for that schema name minus one.

DELete is ON

Turns on an option to automatically delete version 1 of a subschema load module when the subschema is deleted.

DELete is OFF

Turns off an option to automatically delete version 1 of a subschema load module when the subschema is deleted. OFF is the default.

DISplay ALL LIMit is ON

Limits the number of entity occurrences read for a DISPLAY ALL request by the value specified in the INTERRUPT COUNT clause.

DISPLAY ALL LIMit is OFF

Does not limit the number of entity occurrences read for a DISPLAY ALL request. OFF is the default.

ECHo

Specifies that the compiler lists every line it reads (note that lines beginning with *+ are not echoed). Online, input is redisplayed; in batch mode, input appears in the compiler's activity listing.

NO ECHo

Specifies the compiler does not list input lines, whether or not a line contains an error. This option is intended for commands that are submitted 1 line at a time (for example, under TSO local, z/VM local, or from a hard-copy terminal).

EOF is

Designates the 2-character logical end-of-file indicator to be honored by the compiler. When the compiler encounters the indicator coded in the first 2 columns of the input range, it recognizes only the DDL statements that precede the indicator and does not process DDL statements that follow it.

/*

Is the default end-of-file indicator.

'*eof-indicator*'

Is a 2-character value enclosed in quotes.

OFF

Specifies that there is no active end-of-file indicator.

HEAder

(Batch only) Specifies that a heading line identifying the compiler is to appear on the compiler activity listing.

NO HEAder

(Batch only) Specifies that *no* heading line identifying the compiler is to appear on the compiler activity listing.

INPut columns are

Specifies the input range. The compiler reads, in subsequent input lines, only those columns that fall between *start-column-number* and *end-column-number*, inclusive; all other columns are ignored. *Start-column-number* and *end-column-number* must be at least 10 columns apart. The default and maximum ranges depend on the mode in which the compiler is used:

- Online:
  - Full-screen mode (default and maximum) -- 1 through 79
  - Line device (default and maximum) -- 1 through 80
- Batch:
  - Default -- 1 through 72
  - Maximum -- 1 through 80

INTerrupt COUnt is *interrupt-count*

Specifies the number of entity occurrences CA IDMS/DB will read for a DISPLAY ALL request when you specify DISPLAY ALL LIMIT IS ON. *Interrupt-count* is an integer in the range 0 through 32768.

INTerrupt COUnt is NULL

Sets to 0 (zero) the number of entity occurrences CA IDMS/DB will read for a DISPLAY ALL request when you specify DISPLAY ALL LIMIT IS ON. If you attempt to issue a DISPLAY ALL statement when the interrupt count is null (0), CA IDMS/DB will reject the command. NULL is the default.

LISt

Specifies that the compiler lists every line it reads. LIST performs the same function as ECHO.

NO LISt

Specifies that the compiler lists only lines containing errors.

LINes per page is *line-count*

Establishes the number of lines per page for a terminal display or batch activity listing. *Line-count* is an integer in the range 10 through 60. The default is 60.

OUTput line size is

Specifies the width of the terminal display or batch activity listing. The online default is 80; the batch default is 132. Note that with an output line size of 80, error messages do not provide the line numbers of lines in error; the error message, however, will immediately follow the line in error.

user-specification

Establishes the default user for the *user-specification* clause in the SCHEMA and SUBSCHEMA statements and can be overridden in those statements.

Note: Expanded syntax for *user-specification* is presented in Chapter 13, "Parameter Expansions".

If this clause is not used, *user-id* defaults to the user ID known to the DC/UCF system (online compiler) or the user ID known to the batch environment (batch compiler).

PROmpt

Indicates that the compiler will prompt the user for each new line of input when entering DDL source statements line by line (rather than in full-screen mode), as shown in the following example:

ENTER

Note that this option is operational in batch execution, where PROMPT causes the prompt to precede each statement in the compiler's activity listing.

NO PROmpt

Indicates that the compiler will *not* prompt the user for each new line of input when entering DDL source statements line by line (rather than in full-screen mode).

PUNch TO *module-specification*

Specifies that punched output will be directed to the named module in the dictionary. The user can override this default in individual PUNCH statements.

Note: Expanded syntax for *module-specification* is presented in Chapter 13, "Parameter Expansions".

PUNCH to SYSpch

Specifies that punched output will be directed to the system punch file. SYSPCH is the default destination established during installation. The user can override this default in individual PUNCH statements.

QUOte is '/"

Designates a single (') or double (") quote as the quote character in effect for the session. Once set, the selected character must be used in DDL source statements wherever a quotation mark is required.

REGistration OVErride

Turns off schema or subschema security for the session. The user who specifies REGISTRATION OVERRIDE can modify, delete, display, and punch *all* schemas and subschemas, even those whose accessibility otherwise is limited by a PUBLIC ACCESS clause.

SEMicolon alternate end of sentence is ON/OFF

Designates that the schema and subschema compilers will (ON) or will not (OFF) recognize both a semicolon and period as an end of statement terminator. OFF, the default, indicates that the compilers will treat a semicolon as a blank character.

SEQuence is *sequence-number*

Specifies the starting and incremental value for the line numbers to be assigned to record elements and to lines of comment text. *Sequence-number* must be a 1- to 5-digit unsigned integer.

Sequence numbers assigned to record elements are insignificant within the schema compiler itself; however, you can refer to an element by its sequence number when using IDD to modify a record description.

USEr signon OVErride is

Indicates whether CA IDMS/DB will allow users to specify a different user ID in a SIGNON statement from the one known to the environment in which the compiler is executing (the DC/UCF system for online, the batch environment for batch).

ALLowed

Users may sign on to the compiler with a different user ID from the ID known to the execution environment and *user-specification* clauses may be used to override the default user ID. ALLOWED is the default. ON is a synonym for ALLOWED.

NOT ALLowed

CA IDMS/DB will not allow the user ID to be changed. Users who are already known to the environment cannot specify a different user ID in the SIGNON statement. Additionally, *user-specification* clauses cannot be used to change the default user ID. OFF is a synonym for NOT ALLOWED.

DISplay *display-options*

Sets the defaults that govern the output produced by subsequent DISPLAY or PUNCH statements. The defaults established with this clause can be overridden in individual DISPLAY and PUNCH statements.

WITh

Instructs the compiler to include the specified types of information in output produced by DISPLAY/PUNCH statements.

ALSo WITH

Instructs the compiler to include the specified types of information in output produced by DISPLAY/PUNCH statements *in addition* to those currently in effect (either through the SET OPTIONS statement or as set in the individual DISPLAY or PUNCH statement).

WITHOut

Instructs the compiler to *exclude* the specified types of information in output produced by DISPLAY/PUNCH statements.

ALL COMment TYPes

Displays or punches all comment entries (COMMENT, CULPRIT HEADERS, OLQ HEADERS, DEFINITIONS) associated with the schema or subschema.

AREas

Displays or punches all areas in the schema or subschema.

ATTributes

Displays or punches all attributes, and their respective classes, associated with the schema or the subschema.

COMments

Displays or punches comments associated with the schema, schema record, subschema, or logical record.

CULprit headers

Displays or punches all CA Culprit headers for schema elements, when schema record elements are displayed.

DEFinitions

Displays or punches all definitions associated with the subschema.

DETails

Displays or punches details of the component. The details vary depending on the component; they are presented with the syntax for each schema and subschema statement.

ELements

When records for the schema are displayed, displays or punches all elements in COBOL format; when records in the subschema are displayed, all elements included in the subschema definition of the record.

HIStory

Displays or punches the date and time that the schema or subschema was created and/or last modified and the name of the user who created or last modified the schema or subschema.

LRS

Displays or punches all logical records in the subschema.

OLQ headers

Displays or punches all CA OLQ headers for schema elements, when schema record elements are displayed.

PATh-groups

Displays or punches all logical-record path groups in the subschema.

PROgrams

Displays or punches all programs associated with the subschema.

RECords

Displays or punches all database records and elements in the schema or subschema.

SCHemas

Displays or punches the schema related to the displayed or punched schema through the DERIVED FROM option of the SCHEMA statement.

SETs

Displays or punches all sets in the schema or subschema.

SHAred structures

Displays or punches the SHARE STRUCTURE clause of a schema record as syntax and the record's elements as comments.

SUBschemas

Displays or punches all subschemas related to the displayed or punched schema.

SYMbols

Displays or punches all symbols associated with the schema.

SYNonyms

When records for the schema are displayed, displays or punches all record synonyms associated with the schema; when record elements also are displayed, the record and element synonyms associated with the schema.

USEr DEFINED COMments/(UDCS)

Displays or punches all user-defined comment keys associated with the schema and subschema.

USErs

Displays or punches all users associated with the schema or subschema.

ALL

Displays or punches all the information associated with the displayed component. WITH ALL is the default for the DISPLAY clause of the SET OPTIONS statement.

NONE

Displays or punches only the information that uniquely identifies the component: component name; component version, if any; and, for subschemas only, the name and version of the associated schema. Note that NONE is meaningful only when WITH is specified.

VERB

Sets the default for the verb with which the statements are to be produced as the output of DISPLAY and PUNCH statements. For example, if VERB ADD is specified, the output of a later DISPLAY RECORD statement is an ADD RECORD statement; if VERB DELETE is specified, the output of a later DISPLAY RECORD statement is a DELETE RECORD statement; and so on.

The user can override this default in individual DISPLAY and PUNCH statements.

AS COMments

Instructs the compiler to list output produced by a DISPLAY or PUNCH statement in comment format (each line begins with the characters ***+**). These comment characters specify that the line is not to be redisplayed as a function of the ECHO or LIST options.

AS SYNtax

Instructs the compiler to list output produced by a DISPLAY or PUNCH statement in syntax format. Display output AS SYNTAX when you plan to resubmit some or all of the displayed statements to the compiler (for example, when using an existing component description as a template for a new component).

for SESsion

Displays or punches the current options in effect for the session, whether defaulted from installation, set in the dictionary by IDD, or set for the session with the DDL compiler SET OPTIONS statement. FOR SESSION is the default.

for DICtionary

Displays or punches the current options for the dictionary. These options default across sessions. The display does not list options that are only in effect for the session. Dictionary options are set with SET OPTIONS FOR DICTIONARY in the IDD DDDL compiler.

WITh DETails

Specifies that the session or dictionary options are displayed. WITH must be specified to display the options if SET OPTIONS FOR SESSION DISPLAY WITHOUT DETAILS was specified.

WITHOut DETails

Specifies that the session or dictionary options are *not* displayed.

Usage

Schema and Subschema Tasks Performed by DELETE IS ON

In the *subschema compiler*, DELETE IS ON performs the following tasks:

- Deletes version 1 of the subschema load module from the load area of the dictionary when you issue a DELETE SUBSCHEMA command.

  Note: If the subschema load module has a version number other than 1, the load module must be explicitly deleted using the DELETE LOAD MODULE command. For more information about this command, see Chapter 15, “Subschema Statements".

- Erases the PROG-051 dictionary record occurrence associated with the subschema load module, provided the program was built by the subschema compiler and does not participate in any other entity relationships.

In the *schema compiler*, DELETE IS ON performs the same tasks described above for each subschema associated with the schema named in the DELETE SCHEMA command.

Order of Precedence Applied to the LIST and ECHO Options

The LIST and ECHO options have similar functions; the compiler uses the following order of precedence in determining which options will take effect:

1. NO ECHO
2. NO LIST
3. ECHO
4. LIST

This precedence is interpreted as follows: If NO ECHO is set, the setting of LIST or NO LIST is immaterial; if ECHO and NO LIST both are set, NO LIST takes precedence; and so on.

AUTHORITY FOR ALL Required

Only users whose dictionary description specifies AUTHORITY FOR ALL can specify REGISTRATION OVERRIDE or change the following SET OPTIONS settings:

- SIGNON OVERRIDE
- DISPLAY ALL LIMIT
- INTERRUPT COUNT

Other options can be changed by any user holding the necessary authority to use the compiler.

Overriding SET OPTIONS Defaults on Individual Statements

The SET OPTIONS defaults established for user identification, the destination and format of displayed and punched text, and version assignment can be overridden in individual component statements. Other compiler processing options cannot be so overridden and remain in effect until they are reset, either explicitly (by a subsequent SET OPTIONS statement) or automatically.

Options Reset at the Start of Each Session

All options are reset at the beginning of each session:

- **In batch mode**, each time the compiler is executed.
- **Online**, with the first DDL statement issued upon returning to the compiler after either a normal session termination (SIGNOFF) or an abnormal termination of the DC/UCF system. A SIGNON statement that follows session initiation and precedes a SIGNOFF statement (or, in batch mode, the end of the input file) does not begin a new session and, therefore, does not reset all options.

Some Options Reset by the SIGNON Statement

The compiler automatically resets some options to their defaults each time a SIGNON statement is issued. The following table shows which options are reset by the SIGNON statement, which can be changed by the IDD DDDL SET OPTIONS FOR DICTIONARY statement, and the defaults established at installation:

SET OPTIONS option	Installation default	Option Changed by IDD	Option reset by SIGNON
DECIMAL POINT	PERIOD	X	X
DEFAULT	OFF	X	X
DEFAULT FOR EXISTING VERSION	1	X	X
DEFAULT FOR NEW VERSION	1	X	X
DELETE IS ON/OFF	OFF		X
DISPLAY AS	COMMENTS		
DISPLAY ALL LIMIT IS ON/OFF	OFF	X	X
DISPLAY VERB	ADD	X	
DISPLAY WITH	ALL		
ECHO/ NO ECHO	ECHO		
EOF	/*	X	X

SET OPTIONS option	Installation default	Option Changed by IDD	Option reset by SIGNON
HEADER/ NO HEADER	HEADER (batch) NO HEADER (online)		
INPUT COLUMNS	1 THRU 72 (batch) 327 : 1 THRU 79 Line device: 1 THRU 8		
INTERRUPT COUNT IS	NULL	X	X
LINES PER PAGE	60		X
LIST/ NO LIST	LIST		
OUTPUT LINE SIZE	132 (batch) 80 (online)		
PREPARED BY	no default		X
PROMPT/ NO PROMPT	NO PROMPT (batch) 327: NO PROMPT Line device: PROMPT		
PUNCH TO	SYSPCH		
QUOTE	' (single quote)	X	X
REGISTRATION OVERRIDE	OFF		X
REVISED BY	no default		X
SEMICOLON ALTERNATE	OFF	X	X
SEQUENCE	100	X	X
USER SIGNON OVERRIDE	ALLOWED	X	X

DISPLAY/PUNCH Options Valid for Each Compiler

Not all options available for the DISPLAY WITH/ALSO WITH/WITHOUT clause affect all DISPLAY or PUNCH statements. The options that can be specified in this clause apply to DISPLAY or PUNCH statements for specific components, as shown in the following table:

DISPLAY option	Compiler Schema	Compiler Subschema
ALL	X	X
ALL COMMENT TYPES	X	X
AREAS	X	X
ATTRIBUTES	X	X

DISPLAY option	Compiler Schema	Compiler Subschema
COMMENTS	X	X
CULPRIT™ HEADERS	X	
DETAILS	X	X
DEFINITIONS		X
ELEMENTS	X	X
HISTORY	X	X
LRS		X
NONE	X	X
OLQ HEADERS	X	
PATH-GROUPS		X
PROGRAMS		X
RECORDS	X	X
SCHEMAS	X	
SETS	X	X
SHARED STRUCTURES	X	
SUBSCHEMAS	X	
SYNONYMS	X	
USERS	X	X
USER DEFINED COMMENTS		X

Default DISPLAY/PUNCH WITH/WITHOUT DETAILS

The default for WITH/WITHOUT DETAILS on the DISPLAY/PUNCH OPTIONS statement is specified at the session level in the SET OPTIONS statement.

Examples

Sample SET OPTIONS Statement

In this example, the compiler has been instructed to list DISPLAY/PUNCH output in syntax format; each line of input is to be listed; and subsequent input must be specified in the range of columns 2 through 65.

```
set options for session
    display as syntax
    list
    input columns are 2 thru 65.
```

Setting the End-Of-File Indicator

The following example establishes // as the end-of-file indicator for the current compiler session:

```
set options for session
     eof is '//'.
```

More Information

- For more information about modules, see the *CA IDMS IDD DDDL Reference Guide*.
- For more information about assigning authority to users, see the *CA IDMS IDD DDDL Reference Guide*.
- For more information about DISPLAY/PUNCH statement options, see 12.5, "DISPLAY/PUNCH Operations".

SIGNOFF Statement

The SIGNOFF statement signals the end of an online session or batch execution of the schema or subschema compiler, causing the compiler to take the following actions:

- Display a transaction summary
- Free all resources held by the compiler
- Remove the session from the transfer control facility's list of active sessions (if executing under TCF)

Syntax

```
►►─┬─ SIGNOFF ─┬──────────────────────────────────────────────────────►◄
   ├─ BYE ─────┤
   └─ LOGOFF ──┘
```

Usage

Online Use of SIGNOFF

Online, SIGNOFF *does not* transfer control to CA IDMS/DC, DC/UCF, or the transfer control facility; the [Clear] key or the top-line command, [Clear], must follow SIGNOFF in order for the compiler to relinquish control.

When SIGNOFF is Not Required

SIGNOFF is recommended as the best way to terminate a compiler session. However, SIGNOFF is not always required, as described next:

- **Online**, SIGNOFF is required unless the full-screen editor command END is entered. For more information about the END command, see the *CA IDMS Common Facilities Guide*.
- **In batch mode**, SIGNOFF is assumed if the compiler encounters the end of the input file without encountering a SIGNOFF statement.

DDL Compilers Ignore Statements Following SIGNOFF

Any statements following the SIGNOFF command are ignored by the DDL compilers. In the following example, SIGNON and ADD SCHEMA are ignored. To end this session and begin another, eliminating the SIGNOFF statement would produce the desired results.

```
signoff.
signon dictionary=otherdd.
add schema name is othrschm.
```

SIGNON Statement

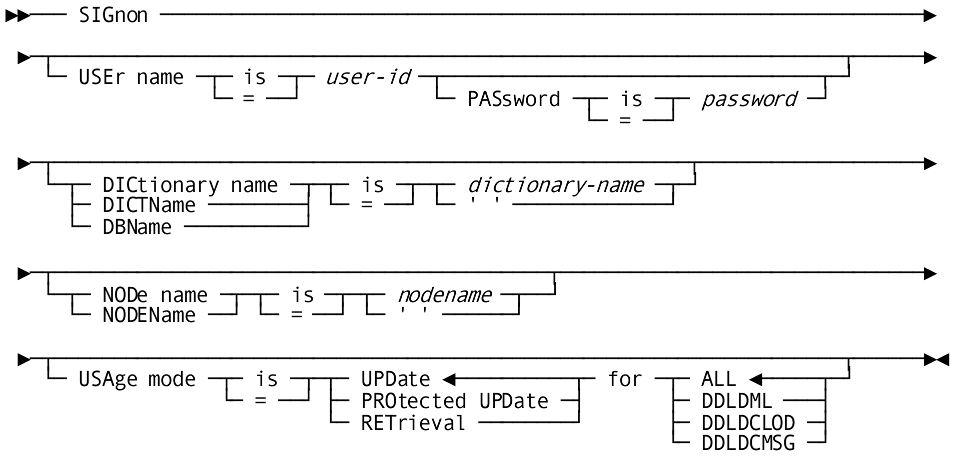
The SIGNON statement permits users to identify themselves to the compiler and to describe the environment in which the compiler is to execute.

Authorization

If IDMS SECURITY is ON in the dictionary, you must already be assigned the appropriate authority (IDMS, SCHEMA, or SUBSCHEMA) through the AUTHORITY clause of the IDD DDDL USER statement.

Note: For more information about the DDDL USER statement, see the *CA IDMS IDD DDDL Reference Guide*.

Syntax



Parameters

USEr name is *user-id*

Specifies the ID of the user signing on to the compiler. If the SECURITY clause of the dictionary (DDDL) SET OPTIONS statement specifies that security for IDMS is on, *user-id* must be the ID of a user authorized (in the DDDL USER clause) for schema or subschema compiler access. *User-id* must be a 1- to 32-character value and must be enclosed in quotation marks if it contains embedded blanks or delimiters.

PASsword is *password*

Specifies the password of the user signing on to the compiler.

DICtionary name is *dictionary-name*

Specifies the dictionary to be accessed by the compiler. If *dictionary-name* is blanks enclosed by quotes, it indicates the default dictionary for the local mode runtime environment or the target node if running under the central version.

NODe name is *nodename*

Specifies the name of the node that controls the dictionary to be accessed. *Nodename* identifies a node in the network. If *nodename* is blanks enclosed in quotes, it indicates the local node (the node at which the online compiler is executing or the DC/UCF system accessed by the batch compiler running under the central version).

USAge mode is

Specifies the manner in which the compiler can access dictionary areas. This clause overrides the usage mode defined during system generation by means of the IDD statement (see the *CA IDMS System Generation Guide*).

UPDate

Specifies that the current user and all other users can update the dictionary concurrently. The compiler automatically prevents deadlock conditions or situations in which users must wait for commands issued by other users to be processed. This is the default, unless overridden during system generation.

PROtected UPDate

Specifies that only the current user can update the dictionary. Other users are restricted to performing retrieval operations. During an online session, the current user has exclusive control for update only if the DDDL compiler has been invoked. Between terminal interactions, the areas can be updated by other users.

RETrieval

Specifies that the current user can only perform retrieval operations against the dictionary. This usage mode does not restrict other users from accessing the dictionary in update or protected update mode.

for ALL

Indicates that the usage mode applies to all areas. ALL is the default.

for DDLDML

Indicates that the usage mode applies only to the DDLDML area.

for DDLDCLOD

Indicates that the usage mode applies only to the DDLDCLOD area.

for DDLDCMSG

Indicates that the usage mode applies only to the DDLDCMSG area.

Usage

When to Specify USER and PASSWORD in SIGNON

If *you are identified to the environment in which the compiler is executing and you do not hold the necessary authorities to perform the intended actions*, you must use the USER clause of SIGNON. In this case, you would specify the ID of a user who holds the necessary authorities (providing USER SIGNON OVERRIDE IS ALLOWED is specified in the SET OPTIONS statement). If the user ID you specify has been assigned a password in the dictionary being accessed, you must also supply that password in the SIGNON statement.

If *you are not identified to the execution environment and IDMS SECURITY is ON*, you must use the USER parameter of SIGNON. In this case, the user ID and password you specify are verified by the central security facility. If verified, you will be known to both the execution environment and the compiler. The user ID must hold the appropriate SCHEMA or SUBSCHEMA authority in the dictionary you are accessing as well as the authority to sign on to the DC/UCF system (if you are executing online). If the user ID you specify has been assigned a password in the central security facility, that password must be specified in the SIGNON statement.

In all other cases, the USER parameter is not required and should not be specified.

Note: For more information about the central security facility, see the *CA IDMS Security Administration Guide*.

Identifying the Dictionary to be Accessed

The DICTIONARY and NODENAME clauses together identify the dictionary to be accessed by the compiler. If only one is specified, the other is derived.

Dictionary-name, if specified, must identify a DBNAME or segment accessible at the target node or local mode runtime environment. If *dictionary-name* is not specified, but *nodename* is specified, then the dictionary is the default dictionary at the specified node.

In local mode, *nodename* has no meaning and is ignored. When running under the central version, *nodename*, if specified, identifies the node at which the target dictionary resides. If not specified, the location of the dictionary is determined from the resource table associated with the local DC/UCF system.

If neither dictionary name nor nodename is specified, they will be established from:

- The TCF specification, if running under TCF
- Session attributes as established by DCUF, SYSIDMS, system or user profiles
- The default dictionary associated with the local runtime environment.

User ID Used in Subsequent DDL Statements

User-id becomes the value assigned in the PREPARED BY and REVISED BY clauses (*user-specification* clause) in subsequent DDL statements, replacing any user named during system signon; this value can be overridden with the SET OPTIONS statement, described in this chapter.

More Information

- For more information about the transfer control facility (TCF), see the *CA IDMS Common Facilities Guide*.
- For more information about DCUF statements, see the *CA IDMS System Tasks and Operator Commands Guide*.
- For more information about dictionary security, see the *CA IDMS IDD DDDL Reference Guide*.
- For more information about central security, see the *CA IDMS Security Administration Guide*.

Chapter 12: Operations on Entities

This section contains the following topics:

ADD Operations

ADD (or the synonym CREATE) does the following:

- Adds schema and subschema entity definitions to the dictionary
- Associates entities with the current schema or subschema

If the Entity Already Exists

If the entity already exists in the dictionary, the response of the compiler depends on the value associated with the DEFAULT clause of the SET OPTIONS statement:

- If DEFAULT IS ON is specified, the compiler interprets the ADD as a MODIFY
- If DEFAULT IS OFF is specified, the compiler issues an error message and terminates processing of the statement.

Defaults

You can explicitly code all characteristics of the added entity or accept one or more default characteristics. Default characteristics are established:

- As dictionary options (using the SET OPTIONS statement)
- As session options (using the SET OPTIONS statement)

The syntax statements identify all default values.

Establishes Update Currency

ADD SCHEMA and ADD SUBSCHEMA statements establish update currency for the specified schema or subschema. Schema or subschema entities can be updated once update currency is established.

Note: For a discussion of currency, see 9.7, “Establishing Schema and Subschema Currency”.

Use VALIDATE After ADD

ADD also sets the schema's or subschema's status to IN ERROR. A VALIDATE statement must set the status to VALID before the schema or subschema becomes a usable component.

MODIFY Operations

MODIFY (or the synonym ALTER) does the following:

- Changes schema and subschema component entity definitions in the dictionary
- Associates component entities with the current schema or subschema

All clauses valid for ADD operations are also valid for MODIFY operations.

Explicitly Code All Changes

All changes to the existing definition must be explicitly coded. Default values apply to ADD operations only.

Establishes Update Currency

MODIFY SCHEMA and MODIFY SUBSCHEMA statements establish update currency for the specified schema or subschema. Schema or subschema component entities can be updated once update currency is established.

Note: For a discussion of currency, see 9.7, “Establishing Schema and Subschema Currency”.

Use VALIDATE After MODIFY

MODIFY also sets the schema's or subschema's status to IN ERROR. A VALIDATE statement must set the status to VALID before the schema or subschema becomes a usable component.

DELETE Operations

DELETE (or the synonym DROP) functions differently for schema and subschema entities. For example, specifying DELETE for a schema area deletes the named area from the dictionary. Specifying DELETE for a subschema area disassociates the named area from the subschema description.

Syntax Presentations Describe Actions

You can find a description of DELETE actions in the detailed syntax descriptions provided for each schema and subschema entity.

VALIDATE Operations

VALIDATE operations cause the schema or subschema compiler to verify the relationships among all components of the schema or subschema that is current for update. Based on this verification, the compiler sets the status to:

- IN ERROR, if it detects errors

  or

- VALID, if it detects no errors

If an error is detected, messages indicate the nature of the error.

Schema and Subschema Status Conditions

The schema or subschema definition in the dictionary carries a status of either IN ERROR or VALID:

- A status of IN ERROR indicates that the definition was not processed by an error-free VALIDATE statement. IN ERROR prevents other CA IDMS/DB software components (for example, a language precompiler) from using the schema or subschema. The schema compiler sets the status to IN ERROR following a successful execution of an ADD or MODIFY SCHEMA statement. Likewise, the subschema compiler does so following ADD or MODIFY SUBSCHEMA.
- A status of VALID indicates that the schema or subschema is usable by other CA IDMS/DB software components. The schema compiler sets the schema's status to VALID after the error-free execution of the VALIDATE statement. Likewise, the subschema compiler does so following the VALIDATE statement or the GENERATE statement.

Use VALIDATE at Any Time During Definition

You can use VALIDATE at any time to verify the relationships of schema or subschema components. For example, you can use VALIDATE when you have not yet defined schema sets, but want to verify the schema's record structures. However, expect a warning for any records whose location mode is VIA an undefined set.

DISPLAY/PUNCH Operations

DISPLAY and PUNCH produce as output the DDL statements that describe the named entity. DISPLAY and PUNCH do not update the entity description.

The location of the output depends on which verb is used and whether the compiler is operating in a batch or online mode:

- *DISPLAY* displays online output at the terminal and lists batch output in the compiler's activity listing.
- *PUNCH* writes the output to the system punch file or to a module in the dictionary. All punched output is also listed in the compiler's activity listing.

Syntax

The following syntax diagram shows the DISPLAY/PUNCH clauses that are common to all DDL entities. Any exceptions are noted in the syntax description for each entity.

Note: For DISPLAY ALL syntax, see Chapter 11, "Compiler-Directive Statements".

```
►►─┬─ DISplay ─┬─ entity-type-name entity-occurrence-name ─────────────────►
   └─ PUNCh ───┘

 ►─┬──────────────────────────┬────────────────────────────────────────────►
   └─ version-specification ─┘

 ►─┬──────────────────────────────────────────────────────┬────────────────►
   └─ PREpared by user-id ─┬─────────────────────────┬──┘
                           └─ PASsword is password ─┘

 ►─┬────────────────────────────────────────────────────────┬──────────────►
   │  ┌──────────────────────────────────────────────────┐ │
   └─▼─┬─ WITh ────────┬─▼─┬────────────────────────────┬─┴─┘
       ├─ ALSo WITh ──┤   └─ entity-option-keyword ───┘
       └─ WITHOut ────┘

 ►─┬─────────────────────────────────────┬─────────────────────────────────►
   └─ TO ─┬─ module-specification ─┬─┘
          └─ SYSpch ───────────────┘

 ►─┬────────────────────────┬──────────────────────────────────────────────►
   └─ VERB ─┬─ ADD ──────┤
            ├─ MODify ───┤
            ├─ DELete ───┤
            ├─ DISplay ──┤
            └─ PUNCh ────┘

 ►─┬───────────────────────┬──────────────────────────────────────────────►◄
   └─ AS ─┬─ COMments ─┬─┘
          └─ SYNtax ───┘
```

Parameters

entity-type-name

Identifies the type of entity to display or punch.

entity-occurrence-name

Specifies the name of the entity occurrence to display or punch. *Entity-occurrence-name* must be the name of an existing occurrence of the specified entity type.

version-specification

Optionally, qualifies the named entity occurrence with a version number. The default is the current session option.

Note: Expanded syntax for *version-specification* is presented in Chapter 13, “Parameter Expansions".

PREpared by *user-id*

Identifies the user who is punching or displaying the entity description. *User-id* can be any 1- to 32-character value; if the value includes spaces or delimiters, it must be enclosed in quotes. The default is the current session option.

If SIGNON OVERRIDE is not allowed, the PREPARED BY clause is ignored and the user is identified as the user known to the runtime.

PASsword is *password*

Supplies the user's password. If *user-id* is assigned a password in the dictionary (through the IDD DDDL compiler), *password* must be that password; if not, the PASSWORD clause is invalid. The default password is the current session option.

WITh

Displays or punches only the parts of the entity description specified by *entity-option-keyword* in addition to parts that always are included such as the entity occurrence name and version. WITH overrides the session defaults specified on the SET OPTIONS statement.

ALSo WITh

Displays or punches the parts of the entity description specified by *entity-option-keyword in addition to* those already in effect (through the SET OPTIONS statement or through the WITH clause in the current DISPLAY statement).

WITHOut

Does *not* display or punch the specified options. *Other* options in effect (through the SET OPTIONS statement or through WITH or ALSO WITH in the current DISPLAY statement) are displayed.

entity-option-keyword

Specifies options to display or punch. *Entity-option-keyword* differs for each entity. See the description of a particular entity for more information.

TO

For PUNCH operations only, specifies the destination of punched output. The default is the current session option.

module-specification

For PUNCH operations only, directs output to the named module in the dictionary.

Note: Expanded syntax for *module-specification* is presented in Chapter 13, “Parameter Expansions".

SYSpch

For PUNCH operations only, directs output to the SYSPCH system punch file

VERB

Specifies the verb with which the entity statement is to be displayed or punched. For example, if VERB ADD is specified, the output of the DISPLAY/PUNCH statement is an ADD statement; if VERB DELETE is specified, the output is a DELETE statement; and so on. If this clause is not coded, the compiler uses the current session option.

AS COMments

Outputs DDL syntax as compiler comments, with *+ preceding the text of the statement. The default is the current session option.

AS SYNtax

Outputs DDL syntax which can be edited and resubmitted to the schema or subschema compiler. The default is the current session option.

Usage

Defaults Determined by SET OPTIONS

DISPLAY and PUNCH default options are determined by the SET OPTIONS statement.

Security Enforcement

If either the compiler or the entity being displayed or punched is secured, the compiler rejects the operation unless the user issuing the statement holds the necessary authority. The user issuing the statement is established by:

- The PREPARED BY clause of the DISPLAY/PUNCH statement
- The *user-specification* in the SET OPTIONS statement
- The user identified in a compiler SIGNON statement
- The user known to the runtime environment in which the compiler is executing

One WITH Clause Per DISPLAY/PUNCH

Only one WITH clause is permitted per DISPLAY/PUNCH operation; if you specify more than one, the compiler applies only the options specified in the last one. To add additional options, use the ALSO WITH option.

Examples

In the following example, the DISPLAY statement includes all current defaults except the schema history.

```
display schema name is empschm
        without history.
```

In the following example, the DISPLAY statement specifies all options (except schema history), whether or not they are included in the current defaults.

```
display schema name is empschm
         with all
         without history.
```

More Information

- For more information about statement syntax, see Chapter 14, “Schema Statements” and Chapter 15, “Subschema Statements”.
- For more information about compiler comments, see Chapter 10, “Using the Schema and Subschema Compilers”.
- For more information about the SET OPTIONS statement and SET OPTIONS session value for user-specification, see Chapter 11, “Compiler-Directive Statements”.

Chapter 13: Parameter Expansions

This section contains the following topics:

Overview

This chapter provides expansions for syntax parameters in other chapters. In a syntax diagram, an expansion is indicated by an underlined and italicized variable. A reference is made from the parameter description to this chapter.

Expansions are shown in alphabetical order, beginning on the next page.

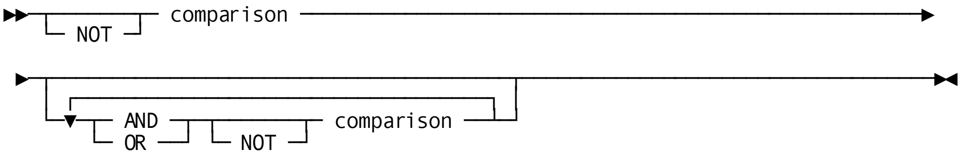
Expansion of boolean-expression

Each FIND/OBTAIN command in a PATH-GROUP statement can include a WHERE clause that specifies boolean selection criteria to be applied to database record occurrences.

The boolean expression can specify as many comparisons as are required to specify the criteria to be applied to the database record. Individual comparisons must be connected by the boolean operators AND, OR, and NOT.

Syntax

Expansion of ***boolean-expression***



Expansion of *comparison*

```
►►─┬─ 'character-string-literal' ─┬───────────────────────────────────────►
   ├─ numeric-literal ────────────┤
   ├─ arithmetic-expression ──────┤
   ├─ db-record-field ──────────┤
   └─ lr-field OF LR ───────────┘

 ►─┬─┬─ EQ ─┬─┬─┬─ 'character-string-literal' ─┬─────────────────────►◄
   │ ├─ IS ─┤ │ ├─ numeric-literal ────────────┤
   │ └─ = ──┘ │ ├─ arithmetic-expression ──────┤
   ├─ NE ─────┤ ├─ db-record-field ────────────┤
   ├─┬─ GT ─┬─┤ └─ lr-field OF LR ─────────────┘
   │ └─ > ──┘ │
   ├─┬─ LT ─┬─┤
   │ └─ < ──┘ │
   ├─ GE ─────┤
   ├─ LE ─────┤
   ├─ CONTAINS ┤
   └─ MATCHES ─┘
```

Parameters

NOT

Specifies that the opposite of the condition fulfills the test requirements.

comparison

'*character-string-literal*'

Specifies an alphanumeric literal enclosed in single quotes.

numeric-literal

Specifies a numeric literal which can be preceded by a minus sign. In numeric literals, if the current decimal point default is a comma, a period (.) is interpreted as an insertion character, and a comma (,) is interpreted as a decimal point.

arithmetic-expression

Specifies an arithmetic expression specified as a minus sign (-), as a simple arithmetic operation, or as a compound arithmetic operation. Arithmetic operators permitted in an arithmetic expression are +, -, *, and /. Operands can be a numeric literal, logical-record field, or database field.

db-record-field

Specifies a data field that participates in the database record named in the path DML command. The field can occur in a record that is accessed but that does not participate in a logical record.

Note: Expanded syntax for *db-record-field* is presented in this chapter.

***lr-field* of LR**

Specifies a data field that participates in the logical record. The OF LR entry is required; it indicates that the value of the named field has been placed in the logical record's variable-storage location by a previous path DML command.

Note: Expanded syntax for *lr-field* is presented in this chapter.

EQ/IS/=

Indicates that the value of the left operand must equal the value of the right operand for the boolean expression to be true. EQ, IS, and = are synonymous.

NE

Indicates that the value of the left operand must not equal the value of the right operand for the boolean expression to be true.

GT/>

Indicates that the value of the left operand must be greater than the value of the right operand for the boolean expression to be true. GT and > are synonymous.

LT/<

Indicates that the value of the left operand must be less than the value of the right operand for the boolean expression to be true. LT and < are synonymous.

GE

Indicates that the value of the left operand must be greater than or equal to the value of the right operand for the boolean expression to be true.

LE

Indicates that the value of the left operand must be less than or equal to the value of the right operand for the boolean expression to be true.

CONTAINS

Indicates that the value of the right operand is contained in the value of the left operand. The value of the right operand must not be longer than the value of the left operand. Note that each operand included with the CONTAINS operator can be a logical-record field name, database record field name, or alphanumeric literal. The fields must be defined as alphanumeric or unsigned zoned decimal values and must be an elementary item.

MATCHES

Indicates that each character in the left operand matches a corresponding character in the right operand (the mask). When MATCHES is specified, CA IDMS/DB compares the left operand with the mask, one character at a time, moving from left to right. The result of the match is either true or false: the result is false if CA IDMS/DB encounters a character in the left operand that does not match the corresponding character in the mask; the result is true if CA IDMS/DB reaches the end of the mask before encountering a character in the left operand that does not match a mask character. Three special characters can be used in the mask to perform pattern matching, as follows:

Special Characters	Description
@	Matches any alphabetic character
#	Matches any numeric character
*	Matches any alphabetic or numeric character

Note that each operand included with the MATCHES operator can be a logical-record field name, database record field name, or alphanumeric literal. The fields must be defined as alphanumeric or unsigned zoned decimal values and must be elementary items.

AND

Indicates the expression is true only if the outcome of both test conditions is true.

OR

Indicates the expression is true if the outcome of either one or both test conditions is true.

Usage

Order of Evaluation

When CA IDMS/DB encounters a boolean expression, it evaluates all operators in the entire boolean expression. Operators are evaluated one at a time, beginning with the operator of the highest precedence. Operators in arithmetic expressions are assigned the highest precedence, followed by comparison operators and boolean operators, respectively. The default order of precedence is shown following:

1. Unary minus in an arithmetic expression (highest precedence)
2. Multiplication and division in an arithmetic expression
3. Addition and subtraction in an arithmetic expression
4. MATCHES and CONTAINS comparison operators
5. EQ, NE, GT, LT, GE, LE comparison operators
6. NOT boolean operator
7. AND boolean operator
8. OR boolean operator (lowest precedence)

Operations of equal precedence are evaluated left to right.

Use Parentheses to Override Default Precedence of Operators

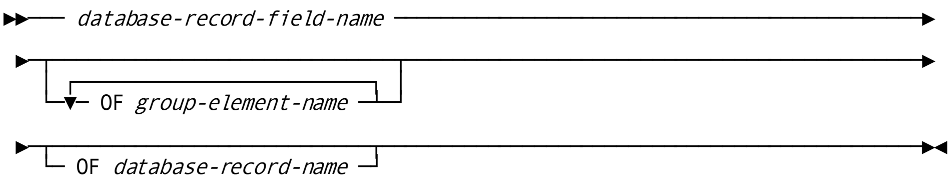
You can use parentheses to override the default precedence of operators and to clarify multiple-comparison boolean expressions. The expression in the innermost parentheses is evaluated first. The keyword NOT can precede a parenthetical expression to negate the result.

Expansion of db-record-field

Db-record-field specifies a data field that participates in the database record named in a PATH GROUP statement.

Syntax

Expansion of ***db-record-field***



Parameters

database-record-field-name

Specifies a data field that participates in the database record named in the path command. If *data-record-field-name* is not unique within the database record named in the path command, at least one of the optional clauses is required.

OF *group-element-name*

Uniquely identifies the named database field. *Group-element-name* names the group element that contains the field. A maximum of 15 different OF *group-element-name* qualifiers can be specified to identify a maximum of 15 levels of group elements.

OF *database-record-name*

Names the database record that contains the field.

Usage

Qualify IDD-Created Synonyms

Note that, although the schema compiler does not allow duplicate elements within a single database record, record synonyms created with IDD can contain such duplicates. Thus, inclusion of such IDD-created synonyms in the subschema can necessitate qualification by group element.

Duplicate Element Names in Records Not Recommended

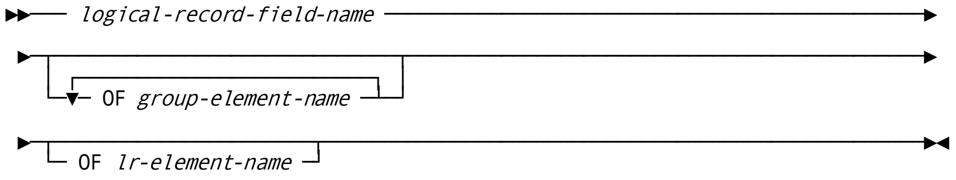
Using duplicate element names in records is not generally recommended because qualification by group element is not supported by CA OLQ, CA Culprit, or navigational DML statements.

Expansion of lr-field

Lr-field specifies a data field that participates in the logical record named in a PATH GROUP statement.

Syntax

Expansion of *lr-field*



Parameters

logical-record-field-name

Specifies a data field that participates in the logical record. If *logical-record-field-name* is not unique within the logical record, code at least one of the optional clauses.

OF *group-element-name*

Uniquely identifies the named database field. *Group-element-name* names the group element that contains the field. A maximum of 15 different OF *group-element-name* qualifiers can be specified to identify up to 15 levels of group elements.

OF *lr-element-name*

Names the logical-record element (database or IDD record) that contains the logical-record field. *Lr-element-name* can be a database record name, an IDD record name, or a role name. If the logical record element containing the logical record field is a record to which a role name has been assigned, *lr-element-name* must be the role name.

Usage

Coding Subscripts for Multiply-Occurring Fields

Code subscripts for multiply-occurring fields after all other qualifiers, including the OF LR and OF REQUEST clauses. For example, to refer to the second occurrence of *logical-record-field-name*, which is defined as occurring three times and which contains a db-key, code the WHERE clause of *find-obtain-dbkey-clause* as follows:

```
WHERE DBKEY = logical-record-field-name OF LR (2)
```

Expansion of module-specification

Module-specification specifies that punched output will be directed to the named module in the dictionary. The named module must exist in the dictionary; the PUNCH function will not create a new module.

Syntax

Expansion of ***module-specification***

```
►►── MODule module-name ─────────────────────────────────────────────►

►─┬───────────────────────┬──────────────────────────────────────────►
  └─ version-specification ┘

►─┬────────────────────────┬─────────────────────────────────────────►
  └─ LANguage is language ─┘

►─┬──────────────────────────────────────────────────────┬──────────►◄
  └─ PREpared by user-id ─┬────────────────────────────┬─┘
                          └─ PASsword is password ─────┘
```

Parameters

MODule *module-name*

Specifies the name of an existing module in the dictionary.

version-specification

Qualifies the named module with a version number. The version number defaults to the current session option for existing versions.

Note: Expanded syntax for *version-specification* is presented in this chapter.

LANguage is *language*

Identifies the language with which the module is associated in the dictionary. If multiple modules with the same name and version number exist in the dictionary, the LANGUAGE clause is required; if the module is not associated with any language, this clause is invalid.

PREpared by *user-id*

Identifies the user who is updating the module. *User-id* can be any 1- to 32-character value; if the value includes spaces or delimiters, it must be enclosed in quotes. The default is the current session option.

PASsword is *password*

Supplies the user's password. If *user-id* is assigned a password in the dictionary (through the IDD DDDL compiler), *password* must be that password; if not, the PASSWORD clause is invalid. The default is the current session option.

Usage

Source Statements Appended to End of Module Source

If the module already contains source statements, the compiler places the punched output at the end of the existing module source; if module source does not exist, the compiler automatically generates a header, which is followed by the punched output. The header contains the date and time that the initial module source was created.

Use PREPARED BY When Compiler Checks Security

PREPARED BY is used when the compiler checks security. If the module is secured, the compiler rejects the operation unless it finds the name and password of an authorized user in one of the following places:

- The PREPARED BY clause of the module specification
- The PREPARED BY clause of the PUNCH statement
- The user identified in the SET options *user-specification*
- The user identified in the signon statement
- The user known to the runtime environment in which the compiler is executing

More Information

- For more information about defining modules, see the *CA IDMS IDD DDDL Reference Guide*.
- For more information about security, see the *CA IDMS IDD DDDL Reference Guide*.

Expansion of user-specification

User-specification identifies the user creating or using the schema entity, subschema entity, or SET OPTIONS statement. This is the user that must hold the authority to perform the operation.

Syntax

Expansion of *user-specification*

```
►►─┬─ PREpared ─┬─ by user-id ─┬────────────────────────┬──────────────────►◄
   └─ REVised ──┘               └─ PASsword is password ─┘
```

Parameters

PREpared/REVised by *user-id*

Identifies the user. *User-id* can be any 1- to 32-character value; if the value includes spaces or delimiters, it must be enclosed in site-standard quotes.

PASsword is *password*

Supplies the user's password. If *user-id* is assigned a password in the dictionary (through the IDD DDDL compiler), *password* must be that password; if not, the PASSWORD clause is invalid.

Usage

Default User-ID

If *user-specification* is omitted from a DDL statement, the user issuing the statement is identified as:

- The user specified in the SET OPTIONS statement
- The user specified in the SIGNON statement
- The user known to the DC/UCF system executing the online compiler or the user known to the batch environment, if executing the batch compiler.

Ignored if SIGNON OVERRIDE NOT ALLOWED

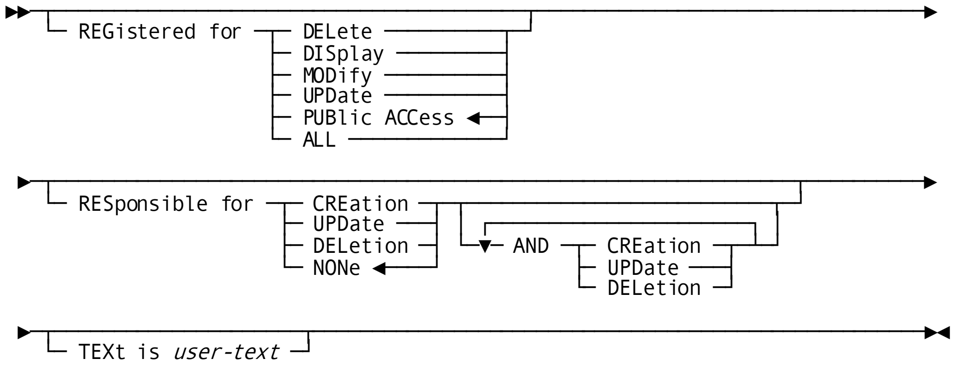
If SIGNON OVERRIDE is not allowed, *user-specification* is ignored and authorization checking is done using the user-id known to the runtime environment.

Expansion of user-options-specification

User-options-specification associates a user with a schema or subschema for security or documentation purposes.

Syntax

Expansion of ***user-options-specification***



Parameters

REGistered for

Authorizes the user to perform the specified types of operations on the schema.

DELete

Allows the user to perform DELETE, DISPLAY, and PUNCH operations only.

DISplay

Allows the user to perform DISPLAY and PUNCH operations only.

MODify

Allows the user to perform MODIFY, DISPLAY, and PUNCH operations only.

UPDate

Allows the user to perform all basic operations: MODIFY, DELETE, DISPLAY, and PUNCH. Unlike ALL, UPDATE neither changes public access nor allows the associated user to change public access.

PUBlic ACCess

Allows the user to perform only those operations, on the schema or subschema, that are available to all users who can sign on to the schema or subschema compiler. PUBLIC ACCESS is the default.

ALL

Allows the user to perform all basic operations: MODIFY, DELETE, DISPLAY, and PUNCH. Additionally, ALL allows the user to issue the PUBLIC ACCESS clause (described in the SCHEMA or SUBSCHEMA statement), thus enabling the user to change security for the schema. If *user-id* is the first user to have this capability, ALL changes public access to NONE.

RESponsible for

Documents a user's responsibility for the schema. It has no effect on the user's authority to access the schema or subschema. Specify any or all of the following options:

- CREATION
- UPDATE
- DELETION
- NONE (default)

TEXt is *user-text*

Allows further documentation of the user's association with the schema or subschema. *User-text* is 1 through 40 characters of text; if it contains spaces or delimiters, it must be enclosed in site-standard quotes.

Note: For more information about the PUBLIC ACCESS authority, see the SCHEMA and SUBSCHEMA statements.

Expansion of version-specification

Version-specification explicitly qualifies an entity with a version number. If you don't specify a version, the default is the current session option for *existing* versions.

Syntax

Expansion of *version-specification*

```
►►─── Version is ─┬─ version-number ─┬───────────────────────────────────►◄
                  ├─ HIGhest ────────┤
                  └─ LOWest ─────────┘
```

Note: NEXT HIGHEST and NEXT LOWEST are options in the VERSION clause of ADD SCHEMA.

Parameters

version-number

Specifies an explicit version number and must be an unsigned integer in the range 1 through 9999.

HIGhest

Specifies the highest version number assigned to the named entity.

LOWest

Specifies the lowest version number assigned to the named entity.

NEXt HIGhest/NEXt LOWest

Establishes the version number of a new schema as the next higher or next lower version with respect to existing schemas with the same name.

Examples

The following ADD SCHEMA statement would assign version 6 to the new schema EMPSCHEM, if version 5 of EMPSCHEM already exists.

```
add schema empschem version next highest.
```

The following is an example of modifying the lowest version. If versions 2, 7, and 11 of schema SOFSCHEM exist in the dictionary, the following statement would cause version 2 of SOFSCHEM to be modified:

```
modify schema sofschem version is lowest.
```

Chapter 14: Schema Statements

This section contains the following topics:

Overview

This chapter describes SCHEMA statements. Syntax, parameter descriptions, usage information, and examples are presented for each statement. Statements are presented in the order in which you use them when you are defining a schema.

Syntax order

ADD/MODIFY syntax is presented first, followed by DELETE syntax. DISPLAY/PUNCH syntax is presented last.

Expansion variables

Diagrams for expansion variables (indicated by underscore and italics) are shown at the end of the current syntax diagram. Expansions for common clauses are handled in a separate chapter, and those expansions are referenced in the parameter description.

Note: For DISPLAY ALL syntax, see Chapter 11, “Compiler-Directive Statements”.

SCHEMA Statement

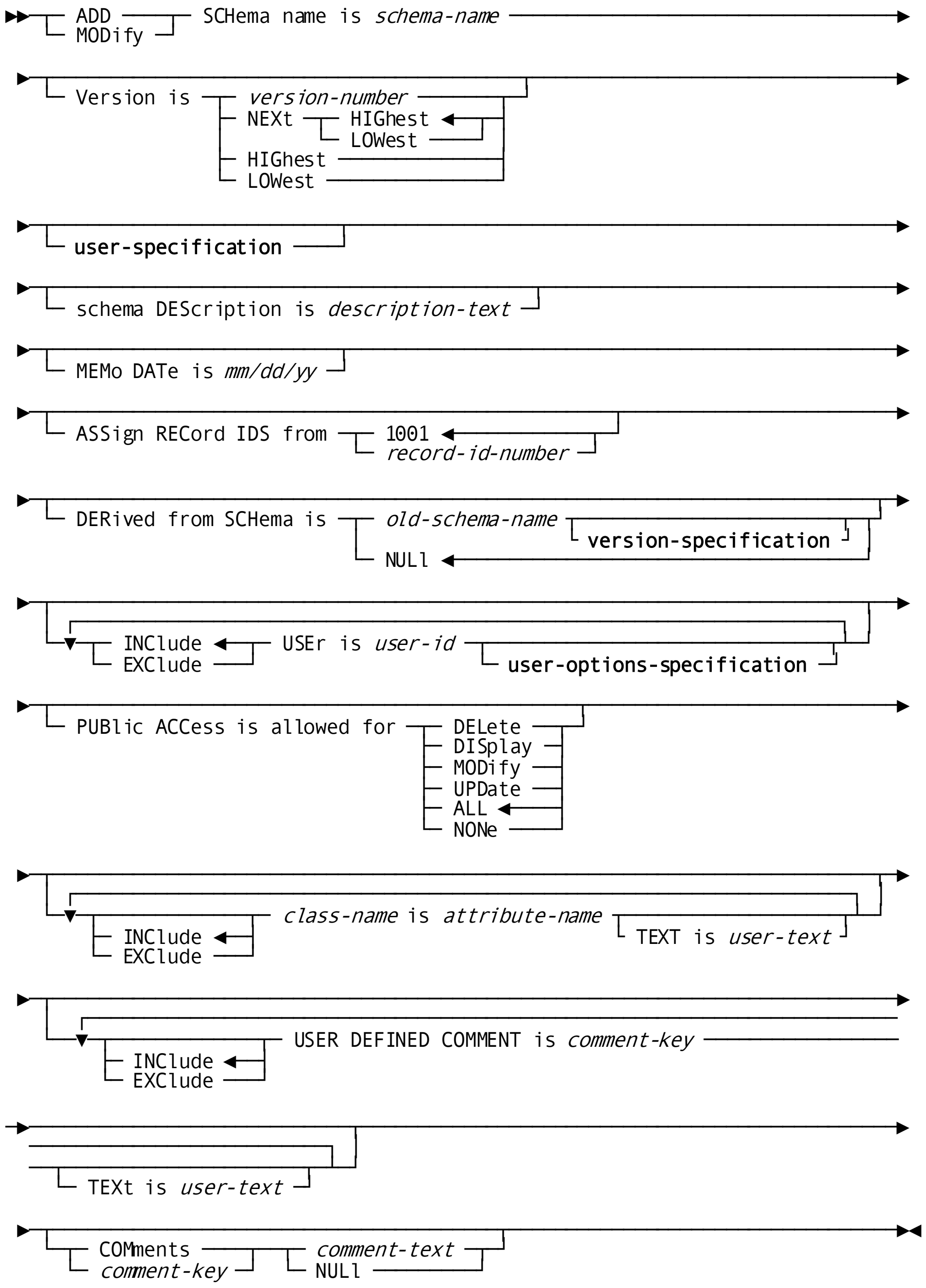
The SCHEMA statements identify the schema as a whole, and establish schema currency as described in .

In addition, SCHEMA statements can:

- Add, modify, delete, display, or punch a schema description
- Establish security for the schema
- Authorize users to issue specific verbs against the schema

Syntax

Syntax: ADD/MODIFY SCHEMA statement



Syntax: DELETE SCHEMA

```
►►─── DELete SCHema name is schema-name ─┬────────────────────────┬───────►
                                         └ version-specification ─┘

►─┬───────────────────────┬────────────────────────────────────────────────►◄
  └ user-specification ─┘
```

Syntax: DISPLAY/PUNCH SCHEMA

```
►►─┬─ DISplay ─┬─ SCHema name is schema-name ─────────────────────────────►
   └─ PUNch ───┘

►─┬──────────────────────────┬─────────────────────────────────────────────►
  └ version-specification ─┘

►─┬───────────────────────────────────────────────────────┬────────────────►
  └ PREpared by user-id ─┬──────────────────────────┬─────┘
                         └ PASsword is password ─┘

►─┬──────────────────────────────────────────────────────────────┬─────────►
  │ ┌────────────────────────────────────────────────────────┐  │
  │ │                  ┌──────────────────────────────────┐  │  │
  └─▼─┬─ WITh ───────┬─▼─┬─ ALL COMment TYPes ─┬────────┴──┴──┘
      ├─ ALSo WITh ─┤   ├─ AREas ──────────────┤
      └─ WITHOut ───┘   ├─ ATTributes ─────────┤
                        ├─ COMments ───────────┤
                        ├─ CULprit headers ────┤
                        ├─ DETails ────────────┤
                        ├─ ELements ───────────┤
                        ├─ HIStory ────────────┤
                        ├─ OLQ headers ────────┤
                        ├─ RECords ────────────┤
                        ├─ SCHemas ────────────┤
                        ├─ SETs ───────────────┤
                        ├─ SHAred structures ──┤
                        ├─ SUBSChemas ─────────┤
                        ├─ SYNonyms ───────────┤
                        ├─ USErs ──────────────┤
                        ├─ ALL ────────────────┤
                        └─ NONe ───────────────┘

►─┬───────────────────────────┬────────────────────────────────────────────►
  └ VERB ─┬─ ADD ──────────┬─┘
          ├─ MODify ──────┤
          ├─ DELete ──────┤
          ├─ DISplay ─────┤
          └─ PUNch ───────┘

►─┬───────────────────────────┬────────────────────────────────────────────►
  └ AS ─┬─ COMments ─────┬──┘
        └─ SYNtax ───────┘

►─┬────────────────────────────────────────────┬───────────────────────────►◄
  └ TO ─┬─ module-specification ─────────┬─────┘
        └─ SYSpch ───────────────────────┘
```

Parameters

SCHema name is *schema-name*

Identifies the schema. *Schema-name* must be a 1- to 8-character value. *Schema-name* must not be the same as any components or synonyms within the schema.

Version is

Qualifies the schema with a version number, which distinguishes this schema from others that have the same name. *Version-number* specifies an explicit version number and must be an unsigned integer in the range 1 through 9999. On an ADD operation, the default is the session default for new versions; on other operations, the default is the session default for existing versions.

NEXt HIGhest

On an ADD operation, specifies the highest version number assigned to *schema-name* plus 1. For example, if versions 3, 5, and 8 of schema CULSCHEM exist in the dictionary, NEXT HIGHEST would define in version 9 of CULSCHEM.

NEXt LOWest

On an ADD operation, specifies the lowest version number assigned to *schema-name* minus 1. For example, if versions 3, 5, and 8 of schema CULSCHEM exist in the dictionary, NEXT LOWEST would define version 2 of CULSCHEM.

HIGhest

On MODIFY and DELETE operations, specifies the highest version number assigned to *schema-name*. For example, if versions 2, 7, and 11 of schema SOFSCHEM exist in the dictionary, HIGHEST would indicate version 11 of SOFSCHM.

LOWest

On MODIFY and DELETE operations, specifies the lowest version number assigned to *schema-name*. For example, if versions 2, 7, and 11 of schema SOFSCHEM exist in the dictionary, LOWEST would indicate version 2 of SOFSCHM.

user-specification

Identifies the user accessing the schema description. If SIGNON OVERRIDE is not allowed, *user-specification* is ignored and the user id identified as the user known to the runtime environment.

Note: Expanded syntax for *user-specification* is presented in Chapter 13, "Parameter Expansions”.

schema DEScription is *description-text*

Optionally specifies a name that is more descriptive than the 8-character schema name required by CA IDMS/DB, but can be used to store *any* type of information. This clause is purely documentational. *Description-text* is a 1- to 40-character alphanumeric field; if it contains spaces or delimiters, it must be enclosed in site-standard quotes.

MEMo DATe is *mm/dd/yy*

Specifies any date the user wishes to supply; it is purely documentational. Note that the time and date of schema creation and last revision are maintained automatically, apart from MEMO DATE, by the schema compiler.

ASSign RECord IDS from *record-id-number*

Specifies the number that the schema compiler will use as a base for numbering schema records. *Record-id-number* must be an unsigned integer in the range 10 through 9999; it defaults to 1001. *Record-id-number* is assigned to the first record in the schema that specifies RECORD ID IS AUTO. the compiler assigns *record-id-number* to that record.

Note: For more information about assigning IDs for subsequent records, see the description of RECORD ID IS AUTO under 14.4, “RECORD Statement”.

DERived from SCHema is *old-schema-name*

Associates the current schema with another schema (*old-schema-name*). This clause is purely informational.

DERived from SCHema is NULl

Dissolves such an association between the current schema and another. It is purely documentational.

INClude USEr is *user-id*

Associates a user with the schema description. *User-id* must be the name of a user as defined in the dictionary.

user-options-specification

Specifies options available to a user associated with the schema.

Note: Expanded syntax for *user-options-specification* is presented in Chapter 13, "Parameter Expansions”.

EXClude USEr is *user-id*

Disassociates a user from the current schema. *User-id* must be the ID of a user as defined in the dictionary.

PUBlic ACCess is allowed for

For the current schema and its components, specifies which operations are available for public access (that is, to all users who can sign on to the schema compiler). When coded, the keyword ALLOWED can be abbreviated to no fewer than 4 characters (ALLO).

DELete

Allows all users to DELETE, DISPLAY, and PUNCH the schema and its components.

DISplay

Allows all users to DISPLAY and PUNCH the schema and its components.

MODify

Allows all users to MODIFY, DISPLAY, and PUNCH the schema and its components.

UPDate

Allows all users to ADD, MODIFY, DELETE, DISPLAY, and PUNCH the schema and its components. Unlike ALL, UPDATE does not allow users to change the schema's PUBLIC ACCESS specification.

ALL

Allows all users to ADD, MODIFY, DELETE, DISPLAY, and PUNCH the schema and its components. Additionally, ALL allows users to change the schema's PUBLIC ACCESS specification, thus enabling them to change security for the schema. ALL is the default.

NONe

Prohibits all users, except those explicitly associated with the schema, from accessing it in any way.

INClude *class-name* is *attribute-name*

Classifies the schema for documentational purposes by associating an attribute with the schema. INCLUDE is the default.

Class-name must be the name of a class as defined in the dictionary through the IDD DDDL compiler. If the dictionary entry for the class specifies that attributes must be added manually, *attribute-name* must be the name of an attribute already associated with *class-name*; if not, *attribute-name* can be any 1- to 40-character value, enclosed in site-standard quotes if it contains spaces or delimiters.

Note: See the *CA IDMS IDD DDDL Reference Guide* for instruction in defining classes and attributes.

TEXT is *user-text*

Supplies additional documentation of the assignment of a specific attribute to the schema. *User-text* is 1 to 40 characters of text; if it contains spaces or delimiters, it must be enclosed in site-standard quotes.

EXClude *class-name* is *attribute-name*

Disassociates an attribute from the schema. *Class-name* must be the name of a class for which an attribute is already associated with the schema; *attribute-name* names the attribute to be disassociated from the schema.

INClude/EXClude USER DEFINED COMMENT is *comment-key*

Identifies a type of comment to be associated with (INCLUDE) or disassociated from (EXCLUDE) the schema. INCLUDE is the default. *Comment-key* must identify an existing user-defined comment type. Values that contain embedded blanks or special characters or that duplicate a keyword from the DDL syntax must be enclosed in site-standard quote characters. Comment text is assigned to the *comment-key* using the COMMENTS clause.

COMments/*comment-key* is *comment-text*/NULl

Updates or removes schema comments. *Comment-key* is the value assigned in the USER DEFINED COMMENTS clause of the IDD DDDL MODIFY ENTITY statement. NULl removes comment text from the current schema.

Note: Coding rules for *comment-text* are presented in 10.5.4, “Coding Comment Text”.

ALL COMment TYPes

Displays and punches all information from the categories COMMENTS, CULPRIT HEADERS, and OLQ HEADERS.

AREas

Displays and punches all areas in the schema.

ATTributes

Displays and punches all attributes, and their respective classes, associated with the schema.

COMments

Displays and punches all comments associated with the schema through the COMMENTS clause of the ADD or MODIFY SCHEMA statement; when RECORDS and ELEMENTS are also specified, all COMMENTS associated with the record elements.

CULprit headers

When RECORDS and ELEMENTS are also specified, displays and punches all CULPRIT HEADERS specified for the record elements.

DETails

Displays and punches information specified previously in the following clauses:

- SCHEMA DESCRIPTION
- MEMO DATE
- ASSIGN RECORD IDS FROM
- PUBLIC ACCESS

ELements

When RECORDS is also specified, displays and punches all elements contained within the records.

HIStory

Displays and punches creation and revision information:

- Creation—The date and time the schema was added to the dictionary and the user who added it (also known as the prepared-by user)
- Revision—The date and time the schema was last modified and the user who modified it (also known as the revised-by user)

OLQ headers

When RECORDS and ELEMENTS are also specified, displays and punches all OLQ HEADERS specified for the record elements.

RECords

Displays and punches all records in the schema, without their associated elements.

SCHemas

Displays and punches the schema associated with the current schema through the DERIVED FROM SCHEMA clause.

SETs

Displays and punches all sets in the schema.

SHAred structures

When RECORDS and DETAILS are also specified, WITH SHARED STRUCTURES displays the SHARE STRUCTURE clause of the record definition as syntax, and the record's elements as comments. WITHOUT SHARED STRUCTURES displays a clause, USES STRUCTURE OF RECORD, as comments, and the record's elements as syntax.

SUBSChemas

Displays and punches all subschemas associated with the schema.

SYNonyms

When RECORDS is also specified, displays and punches the record synonyms associated with the schema; when RECORDS and ELEMENTS are also specified, displays and punches the record and element synonyms associated with the schema.

USErs

Displays and punches all users associated with the schema.

ALL

Displays and punches the entire schema description.

NONe

Displays and punches only the schema name and version number.

Usage

Effect of ADD on Schema

ADD creates a new schema description in the dictionary. Default values established through the SET OPTIONS statement can be used to supplement the user-supplied description.

ADD also sets the schema's status to IN ERROR. A VALIDATE statement must set the status to VALID before a subschema or CA IDMS/DB utility can reference the schema.

Effect of MODIFY on Schema

MODIFY modifies an existing schema description in the data dictionary. This verb also sets the schema's status to IN ERROR. A VALIDATE statement must set the status to VALID before a subschema or CA IDMS/DB utility can reference the schema.

Effect of DELETE on Schema

DELETE deletes an existing schema description and its associated subschema descriptions from the dictionary.

If the SET OPTIONS statement specifies DELETE IS ON, the schema compiler also:

- Logically deletes version 1 of all subschema load modules associated with the schema from the load area of the dictionary (load modules qualified by another version number must be explicitly deleted).
- Automatically erases version 1 of any PROG-051 dictionary record occurrence associated with the subschema load module, provided the record was built by the subschema compiler and is not related to any other entity type in the dictionary.

SCHEMA statement defaults

The schema compiler defaults to supply this information about the schema:

- *Version-number* defaults to the current session option for new versions.
- The record ID assignment begins with 1001.

ADD interpreted as MODIFY

If, on an ADD operation, a schema of the same name and version already exists in the dictionary, the action taken by the schema compiler varies depending on the session option for DEFAULT:

- *If DEFAULT IS ON* was specified, the schema compiler interprets the ADD as a MODIFY for the named schema.
- *If DEFAULT IS OFF* was specified, the schema compiler issues an error message and terminates processing of the ADD SCHEMA statement. Note that, in this case, schema currency will be null for subsequent statements.

Security enforcement

If either authority for SCHEMA is on or the schema being operated on is secured in the dictionary, the user issuing the schema statement must hold the necessary authority to perform the operation. The user issuing the statement is established by:

- *user-specification* in the SCHEMA statement
- *user-specification* in the SET OPTIONS statement
- The user identified in a compiler SIGNON statement
- The user known to the runtime environment in which the compiler is executing

If SIGNON OVERRIDE is not allowed, the user is always the one known to the runtime environment.

USER DEFINED COMMENTS clause

To associate a user-defined comment with a schema:

1. Specify a *comment-key* in the USER DEFINED COMMENTS clause
2. Associate *comment-text* with the key in the COMMENTS clause

If a COMMENTS clause appears in a MODIFY statement, the compiler edits or removes existing comment text.

To remove user-defined comments:

1. Specify NULL in a COMMENTS clause
2. Specify EXCLUDE in a USER DEFINED COMMENTS clause

Use DISPLAY ALL to list all schema names

To list the names of all schemas, issue a DISPLAY ALL statement.

Examples

Minimum SCHEMA statement

The following example supplies the minimum SCHEMA statement required for the purpose of later establishing a functional database:

```
add schema name is sampschm.
```

Using the TEXT clause to document schema revisions

In the following example, the DBA documents schema revisions and the purposes for those revisions; note that the DBA first defined REVISION NUMBER as a class in the dictionary with auto attributes.

```
modify schema name is culschem  version 6
    revision number is '6.5'
        text is 'accommodate new billing procedures'.
```

Note: For more information about the DISPLAY ALL statement, see Chapter 11, “Compiler-Directive Statements".

AREA Statement

The AREA statements identify a logical area of the database. Depending on the verb and options coded, the AREA statements can also:

- Add, modify, delete, display, or punch the area description
- Determine which (if any) database procedures will be executed when the area is accessed at runtime

The schema compiler applies AREA statements to the current schema.

Note: For an explanation of schema currency, see 9.7, “Establishing Schema and Subschema Currency”.

Syntax

Syntax: ADD/MODIFY AREA

```
►►─┬─ ADD ────┬─ AREa name is area-name ─────────────────────────────────►
   └─ MODify ─┘

 ►─┬─────────────────────────────────────────────────────────────────────►─
   └─ SAMe AS area base-area-name ───────────────────────────────────────┘

─►─┬──────────────────────────────────────────────────┬──────────────────►
   └── of SChema base-schema-name ─┬───────────────────────────┬─┘
                                   └─ version-specification ─┘
```

```
►─┬───────────────────────────────────────────────────────────────────────┬─►
  │ ┌─────────────────────────────────────────────────────────────────┐   │
  └─▼─ CALl procedure-name ─┬─ BEFore ─────────────┬─┬───────────────────┬┘───┘
                            ├─ AFTer ──────────────┤ └─ function-option ─┘
                            └─ on ERRor during ────┘

►─┬───────────────────────┬─────────────────────────────────────────────────►
  └─ EXClude ALL CALls ───┘

►─┬──────────────────────────────────────────────────┬──────────────────────►◄
  └─ ESTimated PAGes ─┬─ are ─┬─ page-count ─────────┘
                      └─ is ──┘
```

Expansion of *function-option*

```
►►─┬─ REAdy ─┬────────────────────────────────────────────┬─┬───────────────►◄
   │         └─ for ─┬─ EXCLUSive ─┬─ UPDate ──────┬──────┤ │
   │                 │             └─ RETrieval ───┘      │ │
   │                 ├─ PROtected ─┬─ UPDate ──────┬──────┤ │
   │                 │             └─ RETrieval ───┘      │ │
   │                 ├─ SHAred ─┬─ UPDate ───────┬────────┤ │
   │                 │          └─ RETrieval ────┘        │ │
   │                 ├─ UPDate ───────────────────────────┤ │
   │                 └─ RETrieval ────────────────────────┘ │
   ├─ FINish ───────────────────────────────────────────────┤
   ├─ COMmit ───────────────────────────────────────────────┤
   └─ ROLlback ─────────────────────────────────────────────┘
```

Syntax: DELETE AREA

```
►►─── DELete AREa name is area-name ─────────────────────────────────────────►◄
```

Syntax: DISPLAY/PUNCH AREA

```
►►─┬─ DISplay ─┬─ AREa name is area-name ─────────────────────────────────────►
   └─ PUNch ───┘

►─┬────────────────────────────────────────────────┬──────────────────────────►
  │ ┌──────────────────────────────────────────┐   │
  │ │                       ┌────────────────┐ │   │
  └─▼─┬─ WITh ──────────┬───▼─┬─ DETails ─┬──┴─┴───┘
      ├─ ALSo WITh ─────┤     ├─ ALL ─────┤
      └─ WITHOut ───────┘     └─ NONE ────┘

►─┬─────────────────────────┬─────────────────────────────────────────────────►
  └─ VERB ─┬─ ADD ─────────┬┘
           ├─ MODify ──────┤
           ├─ DELete ──────┤
           ├─ DISplay ─────┤
           └─ PUNch ───────┘

►─┬─────────────────────────┬─────────────────────────────────────────────────►
  └─ AS ─┬─ COMments ──┬────┘
         └─ SYNtax ────┘

►─┬────────────────────────────────────────┬──────────────────────────────────►◄
  └─ TO ─┬─ module-specification ─┬────────┘
         └─ SYSpch ───────────────┘
```

Parameters

AREa name is *area-name*

Identifies the area description. *Area-name* is a 1- to 16-character name that is the same as a physical area name. Apply the following considerations when selecting area names:

- *Area-name* must not be the same as the schema name or the name of any other component (including synonyms) within the schema.
- Because *area-name* will be copied into DML programs, it must not be the name of a keyword known to either the DML precompiler or the host programming language.

SAMe AS area *base-area-name*

Copies the entire area description from an area in another schema into the current schema. *Base-area-name* must identify an existing area.

of *base-schema-name*

Identifies the schema that contains *base-area-name*. The base schema must have a status of VALID (see the VALIDATE statement in this chapter).

version-specification

Qualifies the schema that contains *base-area-name* with a version number. The default version for existing schemas is the current session option.

Note: Expanded syntax for *version-specification* is presented in Chapter 13, “Parameter Expansions".

If the highest version of *base-schema-name* does not contain *base-area-name*, the schema compiler issues an error message; the compiler does not search for the highest schema version that contains *base-area-name*. Likewise, if the lowest version number assigned to *base-schema-name* does not contain *base-area-name*, the schema compiler issues an error message; the compiler does not search for the lowest schema version that contains *base-area-name*.

SAME AS AREA must not be specified for an area to which database procedures already are assigned. Consequently, placement of the SAME AS AREA clause is restricted as follows:

- ADD operation—When used in an ADD operation, SAME AS AREA must precede all other optional clauses.
- MODIFY operation—SAME AS AREA cannot be used in a MODIFY operation unless the area was added with no optional clauses.

As stated earlier, SAME AS AREA copies all information from the copied area to the new area description; the schema compiler treats all subsequent clauses as MODIFY operations.

CALl *procedure-name*

Requests that a system-provided or user-defined database procedure be called at specified times during runtime processing.

Procedure-name is the CSECT name or entry point of an existing procedure. If, at runtime, the procedure is link edited alone for dynamic loading, *procedure-name* must also be the load library member name.

BEFore

Calls a database procedure *before* a runtime READY, FINISH, COMMIT, or ROLLBACK function is performed against the area.

AFTer

Calls a database procedure *after* a runtime READY, FINISH, COMMIT, or ROLLBACK function is performed against the area.

on ERRor during

Calls a database procedure when an error occurs during a runtime READY, FINISH, COMMIT, or ROLLBACK function performed against the area. The DBMS detects an error when the error status is not 0000.

function-option

Specifies the database function that invokes the database procedure. If no function is specified, the procedure is called for every DML function performed against the area.

REAdy

Invokes the database procedure when the runtime system encounters a READY statement.

EXCLUSive

Invokes the database procedure for those runtime READY statements that include either the EXCLUSIVE UPDATE or EXCLUSIVE RETRIEVAL usage mode.

EXCLUSive UPDate

Invokes the database procedure for those runtime READY statements that include the EXCLUSIVE UPDATE usage mode.

EXCLUSive RETrieval

Invokes the database procedure for those runtime READY statements that include the EXCLUSIVE RETRIEVAL usage mode.

PROtected

Invokes the database procedure for those runtime READY statements that include either the PROTECTED UPDATE or PROTECTED RETRIEVAL usage mode.

PROtected UPDate

Invokes the database procedure for those runtime READY statements that include the PROTECTED UPDATE usage mode.

PROTected RETrieval

Invokes the database procedure for those runtime READY statements that include the PROTECTED RETRIEVAL usage mode.

SHAred

Invokes the database procedure for those runtime READY statements that include either the SHARED UPDATE or SHARED RETRIEVAL usage mode.

SHAred UPDate

Invokes the database procedure for those runtime READY statements that include the SHARED RETRIEVAL usage mode.

SHAred RETrieval

Invokes the database procedure for those runtime READY statements that include the SHARED RETRIEVAL usage mode.

UPDate

Invokes the database procedure for those runtime READY statements that include any UPDATE usage mode.

RETrieval

Invokes the database procedure for those runtime READY statements that include any RETRIEVAL usage mode.

FINish

Invokes the database procedure when the runtime system encounters a FINISH statement.

COMmit

Invokes the database procedure when the runtime system encounters a COMMIT statement.

ROLlback

Invokes the database procedure when the runtime system encounters a ROLLBACK statement.

EXClude ALL CALls

Negates any previously assigned CALL clauses for the area.

ESTimated pages is *page-count*

Specifies an estimated page count for the area. *Page-number* is an integer in the range 0 through 1073741822. The default is 0.

Code this option if your transaction performs SQL against a non-SQL database. The value you enter helps the SQL optimizer determine the best way to retrieve records; for example, using an area sweep, an index, and so on.

DETails

Displays or punches with the following information about the area:

- All database procedures assigned to the area
- The type and name of each symbol associated with the area

ALL

Displays or punches the entire area description.

NONe

Displays or punches only the area name.

Usage

DELETE Deletes Area From Subschemas Associated With Schema

DELETE AREA deletes the named area description from the data dictionary. Consequently, the area is removed not only from the current schema, but also from the descriptions of all subschemas associated with the current schema.

SAME AS AREA clause saves coding time

Because SAME AS AREA copies an existing description, it can relieve the DBA of a considerable amount of coding, particularly when many database procedure calls are common across schemas. For an example of assigning database procedures to areas, see the CALL clause, later in this discussion.

You can code multiple CALL statements

Any number of CALL statements for as many DML functions as desired can be specified for an area, as shown in the following example:

```
add area name is ins-prod-region
    same as area ins-demo-region of schema empschm version 1
    call excrash before ready for exclusive
    call securchk before ready for protected update
    call updimsg on error during ready for update
    call countall after finish
    call securlog after ready for update.
```

If more than one BEFORE, AFTER, or ON ERROR procedure is specified for the same function, the procedures are executed in the order specified.

Must respecify all calls to change one call

To change database procedures for an area, *all calls must be respecified*. For example, to remove CALL SECURLOG from the above specification, code the following:

```
mod area name is ins-prod-region
    call countall after finish
    call securchk before ready for update.
    call updimsg on error during ready for update
    call excrash before ready for exclusive.
```

Calls needed for IDMSCOMP compression

If any record in the area uses IDMSCOMP and IDMSDCOM for compression and decompression, the area should have the following database procedure specifications:

```
call idmscomp before finish
call idmsdcom before finish
```

This ensures that the work areas used by the compression and decompression routines are freed when a rununit terminates.

Examples

Sample Minimum AREA Statement

The following example supplies the minimum AREA statement required for the area to be a *valid* schema component:

```
add area name is emp-demo-region.
```

Copying an area from another schema

In the following example, the statement creates the EMP-PROD-REGION area, which is identical to EMP-DEMO-REGION and associates the new area with the current schema:

```
add area name is emp-prod-region
    same as area emp-demo-region
         of schema empschm version is 1.
```

Note: For more information about database procedures, see Chapter 16, “Writing Database Procedures”.

RECORD Statement

The RECORD statements identify a database record type. Depending on the verb, options, and substatements coded, the RECORD statements can also:

- Add, modify, delete, display, or punch the record description
- Assign the record to a database area
- Determine which (if any) database procedures will be executed when occurrences of the record are accessed at runtime
- Create a *record structure*, that is, a dictionary description of the record, including its synonyms, elements, and element synonyms; associate the record with an existing structure

The schema compiler applies RECORD statements to the current schema.

Note: For an explanation of schema currency, see 9.7, "Establishing Schema and Subschema Currency".

Syntax

Syntax ADD/MODIFY RECORD

```
►►─┬─ ADD ────┬─ RECord name is record-name ────────────────────────────►
   └─ MODify ─┘

 ►─┬──────────────────────────────────────────────────────────────────►
   └─ SHAre ─┬─ record-structure-option ───┬┘
             └─ record-description-option ─┘

 ►─┬──────────────────────────────────────────────────────────────────►
   └─ RECord ID is ─┬─ record-id-number ─┬┘
                    └─ AUTo ─────────────┘

 ►─┬──────────────────────────────────────────────────────────────────►
   │ ┌─────────────────────────────────────────────────────┐│
   └─▼─┬───────────────┬── record-synonym-specification ───┴┘
       ├─ INClude ◄────┤
       └─ EXClude ─────┘

 ►─┬──────────────────────────────────────────────────────────────────►
   └─ LOCation MODe is ┬ calc-location-mode-specification ───────────┬┘
                       ├ DIRect ──────────────────────────────────────┤
                       ├ VIA set-name set ┬──────────────────────────┬┤
                       │                  └ displacement-specification ┘│
                       ├ VSAm ────────────────────────────────────────┤
                       └ vsam-calc-location-mode-specification ───────┘
```

```
►─┬────────────────────────────────────────────────────────────┬──────────►
  └─ WIThin AREa area-name ─┬───────────────────────────────────┘
                            ├─ SUBarea symbolic-subarea-name ─┤
                            └─ offset-expression ─────────────┘

►─┬──────────────────────────────────────────────────────────────┬────────►
  └─ VSAm TYPe is ─┬┬─ FIXed ────┬─ LENgth ─┬─ SPAnned ────┬─┤
                   ││└─ VARiable ─┘          └─ NONSPAnned ─┘ │
                   └─ NULl ──────────────────────────────────┘

►─┬──────────────────────────────────────────────────────┬────────────────►
  └─ MINimum ROOT length is ─┬─ root-length characters ─┤
                             ├─ CONtrol length ────────┤
                             ├─ RECord length ─────────┤
                             └─ NULl ──────────────────┘

►─┬────────────────────────────────────────────────────────────┬──────────►
  └─ MINimum FRAgment length is ─┬─ fragment-length characters ─┤
                                 ├─ RECord length ─────────────┤
                                 └─ NULl ──────────────────────┘

   ┌──────────────────────────────────────────────────────────────┐
►──▼─┬──────────────────────────────────────────────────────────────┬────►
     └─ DCTable name ─┬─ BUILTIN ─────┬─┬─────────────────────────────────┤
                      └─ dctable-name ─┘ └─ is used FOR ─┬─ COMpression ───┤
                                                         ├─ DECOMpression ─┤
                                                         └─ BOTh ◄─────────┘

►─┬───────────────────────────────────────────────────────────────────┬───►
  └─ PROcedure name procedure-name is used FOR ─┬─ COMpression ───┤
                                                └─ DECOMpression ─┘

►─┬─────────────────────────────────────────────────────────────────┬─────►
  │ ┌─────────────────────────────────────────────────────────────┐ │
  └─▼─ CALl procedure-name ─┬─ BEFore ──────────┬─┬─ CONnect ────┤─┘
                            ├─ AFTer ───────────┤ ├─ DISCONnect ─┤
                            └─ on ERRor during ─┘ ├─ ERAse ──────┤
                                                  ├─ FINd ───────┤
                                                  ├─ GET ────────┤
                                                  ├─ MODify ─────┤
                                                  └─ STOre ──────┘

►─┬─────────────────────────────────────────────┬─────────────────────────►
  └─ estimated OCCurrences are record-count ─┘

►─┬────────────────────┬──────────────────────────────────────────────────►◄
  └─ EXClude ALL CALls ─┘
```

Expansion of ***record-structure-option***

```
►►─── STRucture of record shared-record-name ─────────────────────────────►

►─┬─────────────────────────────────────────────────────────────┬─────────►◄
  ├─ version-specification ─────────────────────────────────────┤
  └─ of SCHema shared-schema-name ─┬─────────────────────────────┤
                                   └─ version-specification ─┘
```

Expansion of ***record-description-option***

```
►►─── DEScription of record shared-record-name ───────────────────────────►

►─── of SCHema shared-schema-name ────────────────────────────────────────►

►─┬──────────────────────────┬────────────────────────────────────────────►◄
  └─ version-specification ─┘
```

Expansion of *record-synonym-specification*

```
►►── RECord ─┬─ SYNonym name ─┬──────────────────────────────────────────────►
             └─ name SYNonym ─┘

 ►─┬─ IS record-synonym-name FOR language language ─┬────────────────────────►◄
   └─ FOR language language IS record-synonym-name ─┘
```

Expansion of *calc-location-mode-specification*

```
►►── CALc USIng ─┬─ calc-element-name ─────────────┬─────────────────────────►
                 │       ┌─────────────────┐       │
                 └─ ( ─▼─ calc-element-name ─┴─ ) ─┘

 ►── DUPlicates are ─┬─ FIRst ───────┬───────────────────────────────────────►◄
                     ├─ LASt ────────┤
                     ├─ by DBKey ────┤
                     └─ NOT allowed ─┘
```

Expansion of *displacement-specification*

```
►►── DISplacement ─┬─USIng symbolic-displacement-name ─┬─────────────────────►◄
                   └─ page-count pages ────────────────┘
```

Expansion of *vsam-calc-location-mode-specification*

```
►►── VSAm CALc USIng calc-element-name ──────────────────────────────────────►

 ►── DUPlicates are ─┬─ UNORDered ───┬───────────────────────────────────────►◄
                     └─ NOT allowed ─┘
```

Expansion of *offset-expression*

```
►►── OFFset ─┬─ 0 ◄──────────────────────────┬─ for ─┬─ 100 PERcent ◄────┬───►◄
             ├─ offset-page-count PAGes ─────┤       ├─ percent PERcent ─┤
             └─ offset-percent PERcent ──────┘       └─ page-count PAGes ┘
```

Syntax: DELETE RECORD

```
►►── DELete RECord name is record-name ──────────────────────────────────────►◄
```

Syntax: DISPLAY/PUNCH RECORD

```
►►─┬─ DISplay ─┬─ RECord name is record-name ────────────────────────────────►
   └─ PUNch ───┘

 ►─┬──────────────────────────────────────────────────────┬──────────────────►
   │ ┌──────────────────────────────────────────────────┐ │
   │ │                    ┌─────────────────────────┐   │ │
   └─▼─┬─ WITh ──────┬──▼─┬─ ALL COMment TYPes ───┬─┴───┴─┘
       ├─ ALSo WITh ─┤    ├─ AREas ───────────────┤
       └─ WITHOut ───┘    ├─ COMments ────────────┤
                          ├─ CULprit headers ─────┤
                          ├─ DETails ─────────────┤
                          ├─ ELements ────────────┤
                          ├─ OLQ headers ─────────┤
                          ├─ SHAred structures ───┤
                          ├─ SYNonyms ────────────┤
                          ├─ ALL ─────────────────┤
                          └─ NONe ────────────────┘
```

```
►─┬────────────────────────┬───────────────────────────────────────────────►
  └─ VERB ─┬─ ADD ─────────┤
           ├─ MODify ────┤
           ├─ DELete ────┤
           ├─ DISplay ───┤
           └─ PUNch ─────┘

►─┬──────────────────────┬─────────────────────────────────────────────────►
  └─ AS ─┬─ COMments ─┤
         └─ SYNtax ───┘

►─┬──────────────────────────────────┬─────────────────────────────────────►◄
  └─ TO ─┬─ module-specification ─┤
         └─ SYSpch ───────────────┘
```

Parameters

RECord name is *record-name*

Identifies the database record description to be added, modified, or deleted. *Record-name* must be a 1- to 16-character name. The first character must be A through Z (alphabetic), #, $, or @ (international symbols). The remaining characters can be alphabetic or international symbols, 0 through 9, or the hyphen (except as the last character or following another hyphen). *Record-name* must not be the same as the schema name or the name of any other component (including synonyms) within the schema.

SHAre

Connects an existing *record structure* to the schema record. That is, the schema record shares the dictionary description of an existing record, including its synonyms, elements, and element synonyms. Note that, unlike the COPY ELEMENTS substatement the SHARE clause does not create a new record structure.

Note: For more information about contrasting SHARE and COPY ELEMENTS, see "Usage" in this section.

The following considerations apply to the sharing of record structures:

- All schema records that share a single structure must have the same name.
- Any number of identically named records can share a single structure.
- The structure is shared equally among the records; that is, no single record owns the structure.
- When coded, the SHARE clause must precede any RECORD SYNONYM clauses. Synonyms are assigned to the structure and are therefore available to all schema records that share the structure.
- The schema compiler does not allow modification of a shared structure, except to include record synonyms. Nonstructural information (record ID, location mode, and so on) is maintained separately for each schema record and can be modified.
- The SHARE clause and ELEMENT substatements are mutually exclusive. Use SHARE to connect the record to an existing structure; use ELEMENT substatements to create a new structure for the schema record.
- Do not use ELEMENT substatements for any schema record that shares a structure. Once SHAREd, a schema record should always be maintained through SHARE clauses.

record-structure-option

Allows the schema record to share the structure of either a dictionary record (IDD record) or a record that belongs to another schema. The DBA must supply the appropriate RECORD ID, LOCATION MODE, VSAM TYPE, WITHIN AREA, MINIMUM ROOT, MINIMUM FRAGMENT, and CALL clauses, as shown in the following example:

```
add record name is skill
    share structure of record skill
        of schema othrschm
    location mode is calc using skill-code
        duplicates are not allowed
    within area org-demo-region
    minimum root length is control length
    minimum fragment length is record length
    call idmscomp before store
    call idmscomp before modify
    call idmsdcom after get.
```

shared-record-name

Identifies an existing record. While it can be either a primary name or a synonym, *shared-record-name* must be the same as *record-name* (the object of the ADD or MODIFY).

of SCHema *shared-schema-name*

Names the schema associated with *shared-record-name*. *Shared-schema-name* must be the name of a schema, already defined in the dictionary, in which *shared-record-name* participates. The schema must have a status of VALID (see the VALIDATE statement in this chapter).

version-specification

Uniquely qualifies *shared-schema-name* with a version number. The default for existing versions is the current session option.

Note: Expanded syntax for *version-specification* is presented in Chapter 13, "Parameter Expansions".

record-description-option

Allows the schema record to share the structure of a record that belongs to another schema. Unlike SHARE STRUCTURE, SHARE DESCRIPTION copies the remainder of *shared-record-name*'s description (record ID, location mode, and so forth) to the schema record named as the object of the ADD or MODIFY (*record-name*). In the following example, the SKILL record in the current schema shares the structure of the SKILL record in EMPSCHM (version 1); each record has its own copy of nonstructural information:

```
add record name is skill
    share description of record skill
        of schema empschm version 1.
```

SHARE DESCRIPTION is not valid if *record-name* already has nonstructural specifications.

shared-record-name

Identifies an existing record. While it can be either a primary name or a synonym, *shared-record-name* must be the same as *record-name* (named as the object of the ADD or MODIFY). *Shared-record-name must be qualified* with the name of the schema to which it belongs.

version-specification

Uniquely qualifies dictionary records specified for *shared-record-name*. The default is the session option.

Note: Expanded syntax for *version-specification* is presented in Chapter 13, “Parameter Expansions”.

of SCHema *shared-schema-name*

Names the schema associated with *shared-record-name*. This clause is required.

Shared-schema-name must be the name of a schema, already defined in the dictionary, in which *shared-record-name* participates. The schema must have a status of VALID (see 14.8, “VALIDATE Statement”)

version-specification

Uniquely qualifies *shared-schema-name* with a version number. The default for existing versions is the current session option.

Note: Expanded syntax for *version-specification* is presented in Chapter 13, “Parameter Expansions”.

RECord ID is

Assigns a number that uniquely identifies each schema record type. Record IDs are used internally only by CA IDMS/DB software: user-written code never refers to record IDs.

Important! Do not change record IDs for existing databases. Use the RECORD ID clause only when adding new records or when changing records in a schema for which a database is not yet defined.

record-id-number

Specifies an absolute record ID; it must be an unsigned integer in the range 10 through 9999. Record IDs can be duplicated across areas in the schema, however, record IDs must be unique for all records within one area

AUTo

For ADD operations only, indicates that the compiler automatically assigns the record ID. If the record is the first in the schema to be assigned a record ID, AUTO assigns the value specified in the ASSIGN RECORD IDS clause in the SCHEMA statement; otherwise, AUTO assigns a value 1 greater than the highest record ID in the schema, until 9999 is reached. When 9999 is reached, the AUTO attribute assigns the highest unused record ID.

The compiler assigns the ID when the ADD RECORD statement is processed; subsequent displays of the record show the actual ID, rather than the word AUTO.

INClude *record-synonym-specification*

Identifies a record synonym to be associated with the primary record name. A synonym is an alternate name for a record. You can associate more than one record synonym with a record.

record-synonym-name

Names the record synonym. *Record-synonym-name* must follow the rules for the host language with which the synonym is being used and must follow the rules specified above for record names. Record synonyms that will be copied into a subschema or used with OLQ must not exceed 16 characters.

language

Specifies the host language with which the record synonym will be used. Valid values are any of the languages defined in the dictionary, including those defined when CA IDMS/DB is installed: COBOL, PL/I, ASSEMBLER, OLQ, SQL, and CULPRIT. A single synonym may be associated with any number of languages. A record may have only one record synonym associated with language SQL.

You can specify the *language* variable before or after the *record-synonym-name* variable.

EXClude *record-synonym-specification*

Disassociates the named record synonym from the record, provided it is not associated with any other schemas, subschemas, maps, or logical records. If you specify the optional FOR LANGUAGE clause, CA IDMS/DB disassociates the record synonym from the named language.

LOCation MODe is

Defines the technique that CA IDMS/DB will use to physically *store* occurrences of the record type. Each record type must be assigned only one location mode. Note, however, that a record type's location mode does not restrict *retrieval* of record occurrences to a single technique.

calc-location-mode-specification

Specifies that occurrences of the record are to be stored on or near a page that CA IDMS/DB calculates from values in the record element(s) defined by *calc-ele*calc-element-name (the record's CALC key).

calc-element-name

Names any elementary or group data element defined as a record element (see 14.5, “Element Substatement"), with the following restrictions:

- No element named FILLER can be used in the CALC key.
- No repeating element (that is, one defined with an OCCURS clause) and no element subordinate to a repeating element can be used in the CALC key.

Multiple *calc-element-name* values can be coded, forming a compound CALC control element and thereby allowing record placement to be keyed on more than one element within the record. The element names that form the CALC control element need not be contiguous within the member record. The combined lengths of the elements (as defined in the PICTURE and USAGE clauses of the ELEMENT substatement) must not exceed 256 bytes.

If the calc key is to be referenced as a primary key in a set definition, *calc-element-name* must not identify a group element.

DUPlicates are

Specifies whether occurrences of a record type with duplicate CALC key values are allowed and, if allowed, how they are logically positioned relative to the duplicate record already stored.

FIRst

Logically positions record occurrences with a duplicate CALC key before the duplicate record already stored.

LASt

Logically positions record occurrences with a duplicate CALC key after the duplicate record already stored.

by DBKey

Logically positions record(s) occurrences with a duplicate CALC key according to the db-key.

NOT allowed

Indicates that record occurrences with duplicate CALC keys are not allowed.

DIRect

Specifies that occurrences of the record are to be stored on or near a page specified at runtime by the user program.

VIA *set-name* set

Specifies that occurrences of the record are to be stored relative to their owner in a specific set:

- If the member and owner records are assigned to the same page range, the member record occurrences are clustered as close as possible to the owner record.
- If the member and owner records are assigned to different page ranges, the member record occurrences are clustered at locations, within their page range, proportional to the location of the owner within its page range.
- If *set-name* is a system-owned indexed set, CA IDMS/DB will attempt to store the member record in physical sequential order.

Set-name specifies the name of a set in which the record type participates as a member. In most cases, records are defined before sets, so *set-name* need not identify an existing set. However, until the set is defined, the VALIDATE statement will detect errors in the schema.

displacement-specification

Specifies how far away member records are stored from the owner record.

DISplacement USIng *symbolic-displacement-name*

Names a symbol used to represent the displacement. The symbol is assigned a value in a corresponding physical area definition.

DISplacement *page-count* pages

Specifies how far away member records cluster from the owner record when the member and owner record occurrences are assigned to the same page range. The member records cluster starting at the page on which the owner record resides plus *page-count* pages (wrapping around to the beginning of the page range if necessary).

Page-count must be an unsigned integer in the range 0 through 32,767. If *page-count* exceeds the number of pages in the record page range, the displacement wraps around to the beginning of the page range.

VSAm

Specifies that the record is a native VSAM record for which CALC access is required.

vsam-calc-location-mode-specification

Specifies the CALC key used to access occurrences of the record type from a native VSAM file.

USIng *calc-element-name*

Names the element representing the key of a native VSAM file. For KSDS files, *calc-element-name* identifies the primary key; for PATH files, it identifies an alternate index on a KSDS or ESDS file. It also must be defined through an ELEMENT substatement, with the same restrictions as those for the CALC element in the CALC USING clause above.

DUPlicates are

Specifies whether native VSAM record occurrences are allowed to have duplicate CALC keys and if allowed. The DUPLICATES option must correspond to the duplicates option specified when the file was defined to VSAM.

UNORDered

Indicates that CA IDMS/DB stores record occurrences with duplicate CALC keys and *always* retrieves the duplicate record occurrences in the order in which they were stored (whether retrieving forward or backward through the area).

NOT allowed

Indicates that CA IDMS/DB does not store record occurrences with duplicate CALC keys.

WIThin AREa *area-name*

Identifies the area in which occurrences of the record type will be located. *Area-name* must name an area associated with the current schema.

SUBAREA *symbolic-subarea-name*

Names a symbol used to represent a page range (a subarea). Within the physical area definition, the symbolic subarea is assigned the actual range of pages. in which CA IDMS/DB will store occurrences of the record type.

offset-expression

Specifies a relative range of pages, in terms of either a percentage of the physical area or a number of pages in which CA IDMS/DB will store occurrences of the record type. By default, CA IDMS/DB uses the entire physical area.

offset-page-count **PAGES**

Determines the lowest page that CA IDMS/DB should use as the first page to store occurrences of the record type. CA IDMS/DB calculates the actual page, using the formula shown next, when you generate the DMCL that contains the physical area:

```
record's lopage = (LPN + offset-page-count)

     where LPN = the lowest page number in the physical area
```

Offset-page-count must be an integer in the range 0 through the number of pages in *physical-area-name* minus 1.

offset-percent **PERcent**

Determines the first page in which CA IDMS/DB will store occurrences of the record type based on the initial page range of the physical area:

```
record's lopage = (LPN + (INP * offset-percent * .01))

     where LPN = the lowest page number in the physical area
       and INP = the initial number of pages in the physical area
```

Offset-percent must be an integer in the range 0 through 100.

FOR *page-count* PAGes

Determines the last page in which CA IDMS/DB will store occurrences of the record type based on record's low page.

```
record's hipage = (RLP + page-count - 1)

     where RLP = the first page in which occurrences of the
                 record will be stored
```

The calculated page must not exceed the highest page number in the physical area.

FOR *percent* PERcent

Determines the last page in which CA IDMS/DB will store occurrences of the record type based on the record's low page and the total number of pages in the physical area:

```
record's hipage = (RLP + (TNP * percent * .01) - 1)

     where RLP = the first page in which occurrences of the record
                 will be stored
       and TNP = the total number of pages in the physical area
```

Percent must be an integer in the range 1 through 100. The default is 100. If *percent* causes the calculated high page to be greater than the highest page number in the physical area, CA IDMS/DB will ignore the excessive page numbers and will store the record occurrences up to and including the last page in the physical area. The following example is valid and causes EMPLOYEE records to be stored over the last 3/4ths of the area:

```
add record name is employee
    within area emp-demo-region
        offset 25 percent for 100 percent.
```

VSAm TYPe is

Identifies the record as a native VSAM data record and removes or supplies information about how the file containing the record was defined to VSAM. Unless NULL is specified, the options must match those of the VSAM file being described.

For a schema definition to be valid, VSAM TYPE must be supplied for all native VSAM records; this clause is valid only for those records.

FIXed LENgth

Specifies a fixed length record.

VARiable LENgth

Specifies a variable length record.

SPAnned

Specifies that occurrences of the record can span VSAM control intervals.

NONSPAnned

Specifies that occurrences of the record cannot span VSAM control intervals.

NULl

Removes information previously specified in a VSAM TYPE clause.

MINimum ROOT length is

Specifies (or removes the specification for) the minimum portion of a variable-length record that can be stored on a database page. During DML STORE operations, if CA IDMS/DB cannot find a page with enough space to accommodate the minimum root, it will not store the record.

root-length **characters**

Specifies that the initial portion of the record must be the specified number of bytes (characters). *Root-length* must include all CALC, index, and sort control elements. It must be an unsigned integer; if it is not a multiple of 4, the compiler will make it so by rounding up.

CONtrol length

Specifies that the initial portion of the record must include all bytes up to and including the last CALC, index, or sort control element. If the record contains an element defined with an OCCURS DEPENDING ON clause, or if a PROCEDURE NAME or DCTABLE clause is used to indicate compression, CONTROL LENGTH is the default.

RECord length

Specifies that the initial portion of the record must be the entire record (that is, the record is not to be fragmented).

NULl

Removes information previously specified in a MINIMUM ROOT LENGTH clause.

MINimum FRAgment length is

Either specifies the minimum length of subsequent segments (fragments) of a variable-length record or removes such specification (NULL). During DML STORE and MODIFY operations, if CA IDMS/DB cannot find a page with enough space to accommodate the specified portion of the record, it will not store or modify the record.

***fragment-length* characters**

Specifies that subsequent portions of the record must include at least *fragment-length* bytes (an exception is the last fragment, which can be smaller). *Fragment-length* must be an unsigned integer; if it is not a multiple of 4, the compiler will make it so by rounding up. If the record contains an element defined with an OCCURS DEPENDING ON clause, the default is 4.

If the record does not contain an OCCURS DEPENDING ON clause but does contain either a PROCEDURE NAME or a DCTABLE clause (indicating that it is compressed), the default is 40, or (record-length - control-length), whichever is smaller.

RECord length

Specifies that subsequent portions of the record must include the remainder of the record. No more than one fragment will ever be created.

NULl

Removes information previously specified in a MINIMUM FRAGMENT LENGTH clause.

DCTable name

For sites that have installed CA IDMS Presspack, specifies the name of a Data Characteristic Table (DCT). A DCT establishes the best way to compress or decompress records, based upon statistics created by the IDMSPASS utility. This parameter is repeatable so you can specify one DCT for compression and another for decompression.

BUILTIN

Specifies the name of a DCT supplied with CA IDMS Presspack that contains generic information that can be used to compress or decompress any record or set of records.

dctable-name

Specifies the name of a customized DCT. *Dctable-name* is a 1- to 8-character name of a customized DCT created by IDMSPASS.

is used FOR COMPression

Specifies that the named DCT is used to compress records.

is used FOR DECOMpression

Specifies that the named DCT is used to decompress records.

is used FOR BOTh

Specifies that the named DCT is used to compress and decompress records. BOTH is the default.

PROcedure name *procedure-name*

Specifies the name of a standard compression or decompression procedure. *Procedure-name* is the name of a system-provided or user-defined database record compression or decompression procedure. It must be the CSECT name or entry point of an existing procedure. If, at runtime, the procedure is link edited alone for dynamic loading, *procedure-name* must also be the load library member name.

is used FOR COMPression

Specifies that the procedure compresses the record.

is used FOR DECOMpression

Specifies that the procedure decompresses the record.

CALl *procedure-name*

Specifies the name of a system-provided or user-defined database procedure to be called when the runtime system performs the specified DML function against the record. If no function is specified, the procedure is called for every DML function performed against the record.

Procedure-name is the CSECT name or entry point of an existing procedure. If, at runtime, the procedure is link edited alone for dynamic loading, *procedure-name* must also be the load library member name.

If multiple procedures are called for the same function, the procedures are invoked in the order specified.

BEFore

Calls the procedure before the DML function is performed against the record.

AFTer

Calls the procedure after the DML function is performed against the record.

on ERRor during

Calls the procedure when a runtime error occurs during the processing of a DML function against the record. A runtime error exists when the error status is not equal to 0000.

CONnect

Calls the database procedure in response to a CONNECT function.

DISCONnect

Calls the database procedure in response to a DISCONNECT function.

ERAse

Calls the database procedure in response to an ERASE function.

FINd

Calls the database procedure in response to a FIND function. To call a database procedure in response to OBTAIN, code this option and the GET option.

GET

Calls the database procedure in response to a GET function. To call a database procedure in response to OBTAIN, code this option and the FIND option.

MODify

Calls the database procedure in response to a MODIFY function.

STOre

Calls the database procedure in response to a STORE function.

estimated OCCurrences are *record-count*

Specifies an estimated number of record occurrences. CA IDMS/DB uses this value to optimize SQL access to the record. *Record-count* is an integer in the range 0 to 2,147,483,647. The default is 0.

EXClude ALL CALls

Negates any previously assigned CALL clauses for the record.

ALL COMment TYPes

Displays and punches all information from the categories COMMENTS, CULPRIT HEADERS, and OLQ HEADERS.

AREas

Displays and punches the WITHIN AREA clause of the RECORD statement.

COMments

When ELEMENTS is also specified, displays and punches all comments associated with the record elements through the COMMENTS clause of the ELEMENT substatement.

CULprit headers

When ELEMENTS is also specified, displays and punches all CULPRIT HEADERS specified for the record elements.

DETails

Displays and punches the following information about the record:

- The record ID
- The name and version number of the record whose structure was used to create the schema record
- The record's location mode
- The record's VSAM TYPE specification, if any
- The record's MINIMUM ROOT specification
- The record's MINIMUM FRAGMENT specification
- All database procedures assigned to the record

ELements

Displays and punches all elements associated with the record.

OLQ headers

When ELEMENTS is also specified, displays and punches all OLQ HEADERS specified for the record elements.

SHAred structures

When DETAILS is also specified, displays and punches the SHARE STRUCTURE clause of the RECORD statement as syntax and the record's elements as comments; WITHOUT SHARED STRUCTURES displays the USES STRUCTURE clause as comments and the record's elements as syntax.

SYNonyms

Displays and punches the record's synonyms; when ELEMENTS is also specified, the record and element synonyms.

ALL

Displays and punches the entire record description

NONe

Displays and punches only the record name

Usage

Effect of ADD On Records

ADD creates a new schema record description in the data dictionary and associates it with the current schema. The record is known as a *schema record*.

Unless the SHARE clause is used, ADD also creates a record structure for the schema record. The record structure's name is the same as that of the schema record. The structure is automatically assigned a version number, which distinguishes the record from others that have the same name in the dictionary. The schema compiler uses NEXT HIGHEST when assigning record version numbers.

Effect of MODIFY on records

MODIFY modifies an existing schema record in the dictionary. All clauses associated with an ADD operation can be specified for MODIFY operations.

Note: The *CA IDMS IDD DDDL Reference Guide* provides instruction for replacing record elements in schema-owned records under the RECORD entity type discussion. IDD can be used to include documentary information about the record or to modify record elements.

MODIFY operations that affect the record structure

MODIFY operations that use the SHARE clause or ELEMENT substatements affect the record structure. The following considerations apply to such MODIFY operations:

- The SHARE clause and the ELEMENT substatements disassociate the schema record from its existing structure, then associate the record with the specified structure. A schema record's structure is never modified.

  If the disassociated structure becomes *unused* as a result of the MODIFY operation, the schema compiler deletes the structure from the dictionary unless it is used in any of the following ways:

  - Participates in a map
  - Participates in another schema
  - Participates in a subschema logical record
  - Is owned by IDD

  The schema compiler assigns the version number of the unused record to the rebuilt record.

- The schema compiler associates the new record structure with the source of all subschemas that use the record. Subschema load modules, however, must be updated explicitly with a schema REGENERATE or subschema GENERATE statement.

  Note: When a MODIFY operation affects the structure of an existing record, the schema compiler attempts to recreate all partial views of the record, in addition to full views. Subschema views are recreated without view IDs. When a MODIFY operation affects a record used as a logical record element, the logical record must be modified (through the subschema compiler) before the subschema load module can be generated.

Effect of DELETE on records

DELETE operations cause the schema compiler to:

- Remove the named record from both the current schema and the schema's associated subschema descriptions.
- If the DELETE operation causes the record structure to become unused (as described above), delete that structure from the dictionary.
- Delete all sets that the record owns, thus removing such sets from both the current schema and the schema's associated subschema descriptions.
- Remove set membership specifications for all sets in which the record is a member. To delete such a set (if it has no other member records), use the DELETE SET statement.

Defaults supplied on an ADD RECORD statement

The schema compiler defaults supply the following information:

- Record ID is automatically assigned by the compiler
- Record fragmentation (variable-length records only) defaults to a minimum root length of CONTROL LENGTH and to a minimum fragment length of four bytes

Schema record must have at least one element

Every valid schema record must have at least one element (defined in an element substatement) associated with it.

Name records with conventions of programming language in mind

When naming schema records, be sure that the selected names conflict neither with the naming conventions of the programming language(s) that will be used with the CA IDMS Data Manipulation Language (DML) nor with the DML precompilers themselves. As a rule, schema records should bear names that coincide with the language used most often; define record synonyms to accommodate other languages. In addition to the record naming rules stated above, consider the following points when selecting names (or synonyms) for schema records:

- **Assembler** names should not exceed eight bytes in length and should not contain hyphens. When the Assembler DML precompiler (IDMSDMLA) generates a DC or DSECT from a schema record description, it uses the record name as the DC or DSECT name. If the record name exceeds eight bytes in length, IDMSDMLA truncates it, possibly causing duplicate names to appear in the program.
- **COBOL** names must not contain the characters #, $, or @.
- **PL/I** naming conventions coincide with valid CA IDMS/DB schema record names. When the PL/I DML precompiler (IDMSDMLP) generates data field declarations from schema record descriptions, it automatically changes hyphens in the record names to underscores.

Record name for SQL access

If using SQL to access a non-SQL defined database, each record in the non-SQL schema is accessed as a table. The name of the table is always the schema record name (i.e., the object of an ADD RECORD statement). A record synonym for language SQL, if defined, is *not* used as the SQL table name although element synonyms for language SQL are used as column names.

Only one record synonym for language SQL may be defined for a record.

Note: For more information, see the *CA IDMS SQL Reference Guide*.

Considerations for using record synonyms

Record synonyms are language-dependent: each DML precompiler automatically includes the synonym, if any, associated with the compiler-specific programming language (unless instructed otherwise, through a manual COPY or INCLUDE statement).

The following considerations apply when using record synonyms:

- Internally, the schema compiler uniquely identifies record synonyms by assigning version numbers to them.
- The subschema compiler can use any record synonym assigned to a schema record type.
- Record names and synonyms must be unique within a schema. If different schema records have identical synonyms (with different version numbers) in the dictionary, only one such record synonym can be copied into a given schema. *Subschemas* are independent of this restriction.
- Only one record synonym with a language of SQL may exist for a record. This synonym is used when you use SQL to access a non-SQL defined database.
- If a synonym of a schema record is associated with the language of a program being precompiled, the precompiler copies that synonym instead of the schema's primary record (unless instructed otherwise, through a manual COPY or INCLUDE statement).
- If the record copied into a program by a DML precompiler is a synonym of a schema record, the DML precompiler treats the synonym as if it were the schema record (for example, in the BIND statement).

When to use area page counts

Use area page counts (for example, the OFFSET *page-count* clause) under these conditions:

- For records accessed by an area sweep (for example, DIRECT records)
- To exclude an area's SMP page (if the area has only 1), from the CALC algorithm

Differences between SHARE STRUCTURE/DESCRIPTION clauses

Both **SHARE STRUCTURE** and **SHARE DESCRIPTION** cause the schema record to share the structure of an existing record. The differences between the two are:

- SHARE DESCRIPTION shares the structure of another schema record; SHARE STRUCTURE shares the structure of either a dictionary record (IDD record) or another schema record.
- SHARE DESCRIPTION additionally copies the nonstructural part of the existing schema record; SHARE STRUCTURE does not. The nonstructural part is the record ID, location mode, VSAM type, area, minimum root length, minimum fragment length, and database procedures associated with the schema record.

SHARE DESCRIPTION must appear first

SHARE DESCRIPTION must be the first clause in the ADD or MODIFY RECORD statement. Any clauses that follow SHARE DESCRIPTION are applied to the record description as modifications. Thus, the DBA can share the description of a record that is similar to the one needed, and code only those clauses that represent differences between the two records. For this usage, select as *shared-record-name* a record whose structure is identical to that needed: while the descriptive part of the record can be changed directly, the structural part cannot.

Percentage offsets provide most flexibility

Of the page limiting options, OFFSET with percentage specifications is the most flexible. As a database grows and must eventually be expanded, the physical areas of the database must also be expanded. If the DBA originally expresses a record type's page range as a percentage of a physical area, the schema compiler "remembers" the percentage. Consequently, when the physical area is later expanded, the DBA need not respecify the record's page range; the schema compiler will automatically assign the record type to the appropriate percentage of the new physical area.

MINIMUM ROOT/FRAGMENT clauses can apply to fixed-length records

The MINIMUM ROOT LENGTH and MINIMUM FRAGMENT LENGTH clauses also apply to fixed-length records if those records are being processed by the compression (IDMSCOMP) and decompression (IDMSDCOM) procedures or IDMS/Presspack. The schema compiler automatically generates these clauses in response to a PROCEDURE NAME or DCTABLE NAME clause.

The MINIMUM ROOT LENGTH and MINIMUM FRAGMENT LENGTH clauses are allowable, but not functional, for native VSAM records.

Storing variable-length records

CA IDMS/DB never stores a variable-length record on a page unless sufficient space exists for the minimum root, and it never stores fragments smaller than the specified minimum, except for the last fragment. If MINIMUM ROOT LENGTH IS RECORD LENGTH is specified and a record occurrence is larger than page size minus 40, CA IDMS/DB returns an error-status code of 1211 (on a STORE) or 0811 (on a MODIFY). The same is true if MINIMUM FRAGMENT LENGTH IS RECORD LENGTH is specified and a record fragment is larger than page size minus 40.

Modifying the record size can cause fragmentation

Increasing the size of the record occurrence with a runtime MODIFY operation can necessitate fragmentation even though fragmentation was not specified in the schema. For example, if MINIMUM ROOT LENGTH IS RECORD LENGTH, CA IDMS/DB stores the current length of the record without fragmentation. However, if a record occurrence is later modified to a length that exceeds available page space, it may be fragmented at that time. Similarly, if MINIMUM FRAGMENT LENGTH IS RECORD LENGTH is specified, no occurrence of the record is fragmented more than once (root plus 1 fragment) upon storage, but an occurrence can be further fragmented if its length is increased as a result of modification.

Considerations for variable-length/-compressed records

For variable and variable-compressed record types, both the MINIMUM ROOT LENGTH and MINIMUM FRAGMENT LENGTH clauses can be omitted and the defaults taken; CA IDMS/DB recognizes the record as variable from the OCCURS DEPENDING ON clause in a record element description (see 14.5, “Element Substatement”).

Note: For documentation purposes, the best practice is to always include both the MINIMUM ROOT LENGTH and MINIMUM FRAGMENT LENGTH clauses in the description of all variable-length records if you specify CALL statements for record compression and decompression.

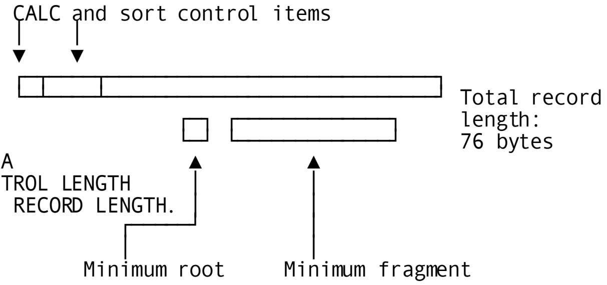
Considerations for fixed-compressed record types

For fixed-compressed record types, MINIMUM ROOT LENGTH IS CONTROL LENGTH must be specified explicitly if you use CALL statements to specify IDMSCOMP and IDMSDCOM for compression and decompression. This will ensure the proper result from the IDMSCOMP and IDMSDCOM procedures. If neither of the two clauses is specified, the compression procedures will compress data; however, the record will consume its full, fixed length in storage.

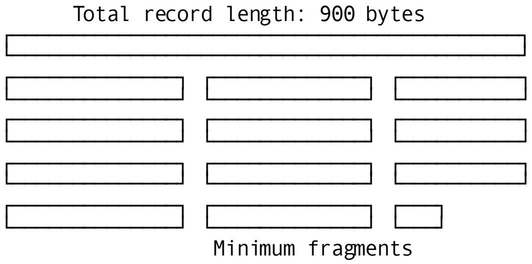
MINIMUM ROOT and MINIMUM FRAGMENT examples

EXAMPLE 1:

```
ADD RECORD NAME IS SKILL
  LOCATION MODE IS CALC
     USING SKILL-CODE
  DUPLICATES NOT ALLOWED
  WITHIN ORG-DEMO-REGION AREA
  MINIMUM ROOT LENGTH IS CONTROL LENGTH
  MINIMUM FRAGMENT LENGTH IS RECORD LENGTH.
```



```
EXAMPLE 2:


ADD RECORD NAME IS DENTAL-CLAIM
   LOCATION MODE IS VIA
     COVERAGE-CLAIMS SET
   WITHIN INS-DEMO-REGION AREA
   MINIMUM ROOT LENGTH IS 0
   MINIMUM FRAGMENT LENGTH IS 80.
```



Minimum root and fragment lengths assigned to compressed records

The schema compiler assigns the following minimum root and fragment lengths to the record definition when it processes a DCTABLE clause or a PROCEDURE NAME clause:

- MINIMUM ROOT LENGTH IS CONTROL LENGTH
- MINIMUM FRAGMENT LENGTH IS 4 for variable compressed records
- MINIMUM FRAGMENT LENGTH IS the lesser of 40 and (record-length - control-length) for fixed compressed records

You can override the defaults by explicitly coding the MINIMUM ROOT and MINIMUM FRAGMENT clauses.

Respecify procedure statements if procedure is updated/deleted

If you want to add, modify, or delete a DCT, a standard compression or decompression procedure or other CALL procedures for a record, all DCTABLE, PROCEDURE, and CALL clauses must be respecified when you modify the record.

Area procedures needed for IDMSCOMP compression

If any record in the area uses IDMSCOMP and IDMSDCOM for compression and decompression, the area should have the following database procedure specifications:

```
CALL IDMSCOMP BEFORE FINISH.
CALL IDMSCOMP BEFORE ROLLBACK.
CALL IDMSDCOM BEFORE FINISH.
CALL IDMSDCOM BEFORE ROLLBACK.
```

This ensures that the work areas used by the compression and decompression routines are freed when a rununit terminates.

Implied CALL statements generated by PROCEDURE NAME

The PROCEDURE NAME clause generates the equivalent of the following CALL statements, depending on whether the clause specifies COMPRESSION or DECOMPRESSION:

Procedure	CALL statements
COMPRESSION	CALL procedure-name BEFORE MODIFY CALL procedure-name BEFORE STORE
DECOMPRESSION	CALL procedure-name AFTER GET

Code as many CALL clauses as necessary

Any number of CALL clauses for as many DML functions as necessary can be specified for a record, as shown in the following example. If more than one BEFORE, AFTER, or ERROR procedure is specified for the same function, the procedures are executed in the order specified.

```
add record name is insurance-plan
    location mode is calc using code
        duplicates are not allowed
    within area ins-demo-region
    call inrecs after get
    call error-check after get.
```

Examples

Minimum RECORD statement for an uncompressed record

The following example supplies the minimum RECORD statement required for an uncompressed record to be a *valid* schema component:

```
add record name is employee
    location mode is calc using emp-id
        duplicates are not allowed
    within area emp-demo-region.
    02  emp-id pic xxxx.
```

Minimum RECORD statement for a fixed-length compressed record

The following example supplies the minimum RECORD statement required for a fixed-length, compressed record to be a *valid* schema component:

```
add record name is job
    location mode is calc using job-id
        duplicates are not allowed
    within area org-demo-region
    procedure name idmscomp is used for compression
    procedure name idmsdcom is used for decompression
    02  job-id pic xxxx.
```

The above example specifies procedures to call to compress and decompress the JOB record. By default, the schema compiler supplies a minimum root length and fragment length for the record. Note that you can also compress and decompress a record by using CA IDMS Presspack.

Specifying a record synonym

The following example specifies a synonym for a record named DENTAL-CLAIM to be used by an Assembler program (for which record names must not be longer than eight characters):

```
add record name is dental-claim
    location mode is via coverage-claims set
    within ins-demo-region area
    record synonym name for assembler is dntlclm.
```

Specifying an area percentage offset

Logical area, EMP-DEMO-REGION, has been defined to physical areas within the PROD and TEST segments. PROD.EMP-DEMO-REGION contains 1000 pages, numbered 1 through 1000, with an additional 500 pages (1001 through 1500) reserved for extending the physical area. TEST.EMP-DEMO-REGION contains 100 pages, numbered 1501 through 1600.

Record EMPLOYEE is defined to EMP-DEMO-REGION as follows:

```
add record name is employee
  within area emp-demo-region
  offset 25 percent for 75 percent.
```

Using the percentage offset specified for the EMPLOYEE record, the runtime system calculates the low and high pages for the record in the initial page range of PROD.EMP-DEMO-REGION:

- Low page is 251 (1 + (1000 * 25 * .01))
- High page is 1000 (251 + (1000 * 75 * .01) -1).

For TEST.EMP-DEMO-REGION, the first and last usable page is:

- Low page is 1526 (1501 + (100 * 25 * .01)
- High page is 1600 (1526 + (100 * 75 * .01) -1).

When you extend PROD.EMP-DEMO-REGION by 500 pages (page 1 through 1500) using the percentage offsets specified for the EMPLOYEE record, the runtime system calculates the record's low and high pages in the extended page range:

- Low page is 251 (1 + (1000 * 25 * .01)).
- High page is 1375 (251 + (1500 * 75 * .01) - 1).

CA IDMS/DB will store occurrences of the EMPLOYEE record on pages numbered 251 through 1375 in the PROD.EMP-DEMO-REGION area. If the record's location mode is CALC, the record will continue to target to its initial page range of 251 through 1000 and overflow, if necessary, into the extended pages 1001 through 1375.

Specifying a relative page offset for an area

In the following example, physical area ORG-DEMO-REGION in segment PROD contains 240 pages, numbered from 2001 through 2240. The schema description of the DEPARTMENT record is:

```
add record name is department
    within area org-demo-region
        offset 2 pages for 238 pages.
```

Using the offset specified for the DEPARTMENT record, the runtime system calculates the low and high pages for the record as:

- The low page is 2003 (2001 + 2).
- The high page is 2240 (2003 + 238 - 1).

CA IDMS/DB will store occurrences of the DEPARTMENT record on pages numbered 2003 through 2240 in the PROD.ORG-DEMO-REGION area.

Modifying a record by adding new routines

In the next example, schema record EMPREC is modified by adding two routines to handle errors that occur when a record is obtained or stored. The code must respecify the PROCEDURE name clauses for the standard compression and decompression routines because of the new CALL clauses.

```
modify record name is emprec
  procedure name is idmscomp is used for compression
  procedure name is idmsdcom is used for decompression
  call errrtn on error during store
  call errget on error during get.
```

More Information

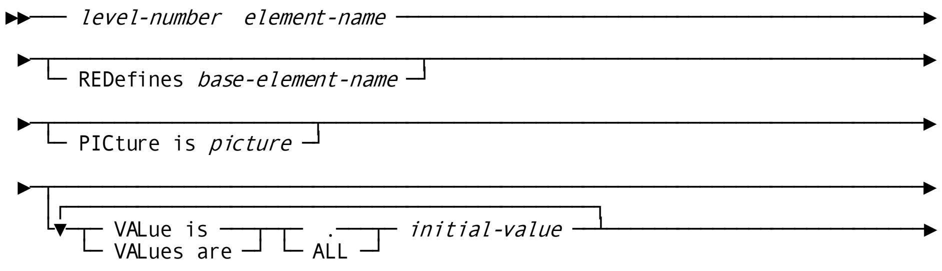
- For more information about database procedures, see Chapter 16, “Writing Database Procedures”.
- For more information about variable-length records and how they are stored, see Chapter 36, "Record Storage and Deletion".
- For more information about CA IDMS Presspack, see the *CA IDMS Presspack User Guide*.

Element Substatement

The element substatement associates an element with the record and, if the element does not already exist, adds the element description to the dictionary. Schema element descriptions cannot be modified or deleted. To change element descriptions, modify the record description and respecify all of the record's elements.

Syntax

Element substatement



```
 ►─┬───────────────────────────────────────────────────────────────────────►
   │ ┌───────────────────────────────────────────────────────────────────┐
   └─▼─┬─ VALue is ──┬──────  . ─────────────────────────────────────────────
       └─ VALues are ┘

►──────────────────────────────────────────────────────────────────────────┬─►
    ┌──────────────────────────────────────────────────────────────────┐  │
 ───┬─┬───────┬─ condition-value ─────────────────────────────────────┬─┘
    │ └─ ALL ─┘                                                        │
    │   ┌──────────────────────────────────────────────────────────┐  │
    └─ (▼─┬───────┬─ condition-value ─┬─────────────────────────────────────┬─┴─ ) ─┘
          └─ ALL ─┘                   └─ THRu ─┬───────┬─ condition-value ─┘
                                               └─ ALL ─┘

 ►─┬─────────────────────────────────────────────────────────────────────────►
   └─ USAge is ─┬─ BIT ─────────────────────┬─┘
                ├─┬─ COMPUTATIONAL ──────┬──┤
                │ ├─ COMp ───────────────┤  │
                │ └─ BINary ─────────────┘  │
                ├─┬─ COMPUTATIONAL-1 ──┬────┤
                │ ├─ COMP-1 ───────────┤    │
                │ └─ SHOrt-point ──────┘    │
                ├─┬─ COMPUTATIONAL-2 ──┬────┤
                │ ├─ COMP-2 ───────────┤    │
                │ └─ LONg-point ───────┘    │
                ├─┬─ COMPUTATIONAL-3 ──┬────┤
                │ ├─ COMP-3 ───────────┤    │
                │ └─ PACked ───────────┘    │
                ├─┬─ COMPUTATIONAL-4 ──┬────┤
                │ └─ COMP-4 ───────────┘    │
                ├─ CONdition-name ──────────┤
                ├─ DISplay ─────────────────┤
                ├─ DISplay-1 ───────────────┤
                └─ POInter ─────────────────┘

 ►─┬─────────────────────────────────────────────────────────────────────────►
   └─ SYNChronized ─┬──────────┬─┘
                    ├─ LEFt ───┤
                    └─ RIGht ──┘

 ►─┬───────────────────────────────────────────────────────────────────────────►─
   └─ OCCurs ───────────────────────────────────────────────────────────────────

►─────────────────────────────────────────────────────────────────────────────┬─►
 ─┬─ occurrence-count times ──────────────────────────────────────────────────┘
  └─┬─ occurrence-count ────────┬─ times DEPending on control-element-name ─┘
    └─ 0 TO occurrence-count ───┘

 ►─┬─────────────────────────────────────────────────────────────────────────►
   └─ JUStify RIGht ─┘

 ►─┬─────────────────────────────────────────────────────────────────────────►
   └─ BLAnk when ZERo ─┘

 ►─┬─────────────────────────────────────────────────────────────────────────►
   └─ SIGn is ─┬─ LEAding ──┬─┬────────────────────────┬─┘
               └─ TRAiling ─┘ └─ SEParate character ─┘

 ►─┬─────────────────────────────────────────────────────────────────────────►
   │ ┌────────────────────────────────────────┐ │
   └─▼─ element-synonym-specification ──────┴─┘

 ►─┬─────────────────────────────────────────────────────────────────────────►
   └─ INDexed BY ─┬─ index-name ─────────────────────┬─┘
                  │      ┌────────────────┐          │
                  └─ ( ──▼─ index-name ───┴── ) ─────┘
```

```
►─┬──────────────────────────────────────────────────────────────────────────┬──►
  ├─ INDex KEY is ─┬─ index-name ─┬─ ASCending ──┬────────────────────────┐ │
  │                │              └─ DEScending ─┘                        │ │
  │                │     ┌──────────────────────────────────────┐          │ │
  │                └─ ( ─▼─ index-name ─┬─ ASCending ──┬────────┴─ ) ─────┘ │
  │                                     └─ DEScending ─┘                   │
  └─┬─ ASCending ──┬─ key is ─┬─ index-name ───────────────┬───────────────┘
    └─ DEScending ─┘          │        ┌─────────────┐     │
                              └─ ( ───▼─ index-name ─┴─ ) ─┘

►─┬──────────────────────────────────────────────────────────────────────────┬──►
  │ ┌──────────────────────────────────────────────────────────────────────┐ │
  └─▼─ EDIt ─┬─────────────┬─ TABle is ( ─▼─ 'value' ─┬────────────────┬─┴─ ) ─┘
             ├─ VALid ◄────┤              │           └─ THRu 'value' ─┘ │
             └─ INValid ───┘              └──────────────────────────────┘

►─┬──────────────────────────────────────────────────────────────────────┬──►
  │ ┌────────────────────────────────────────────────────────────────┐   │
  │ │        ┌─────────────────────────────────────────────┐         │   │
  └─▼─ CODe TABle is ( ─▼─ 'encode-value' 'decode-value' ──┴─── ) ───┴───┘

►─┬──────────────────────────────────────┬──►
  └─ EXTernal PICture is picture ────────┘

►─┬────────────────────────────────────────────────────────────────┬──►◄
  │┌──────────────────────────────────────────────────────────┐    │
  └▼─┬─ OLQ header ──────┬─ is ─┬─ 'comment-text' ─┬──────────┴────┘
     ├─ CULprit header ──┤      └─ NULl ───────────┘
     ├─ COMments ────────┤
     ├─ DEFinitions ─────┤
     └─ comment-key ─────┘
```

Expansion of *element-synonym-specification*

```
►►── element ─┬─ SYNonym name ─┬──────────────────────────────────────►
              └─ name SYNonym ─┘

►─── FOR language language is synonym-name ───────────────────────────►◄
```

Parameters

level-number

Indicates the level within the record to be occupied by the element. The level number must be an unsigned integer in the range 02 through 49, or 88. Level 88 applies to records used with CA ADS or COBOL only. Note that the highest level (01) in any record description is assigned by CA IDMS/DB to the record itself. The COBOL and PL/I DML precompilers can be directed to change the level numbers when the record is copied into a program (see the language-specific CA IDMS DML reference).

element-name

Identifies the element to be added to the record description. *Element-name* must be a 1- to 32-character name. The first character must be A through Z (alphabetic), digit (0 through 9), #, $, or @ (international symbols). The hyphen can also be used except as the first or last character, or following another hyphen. *Element-name* must not be the same as the schema name or the name of any other component (including synonyms) within the schema, with the following exceptions:

- An element name or synonym can be duplicated within a schema, but must be unique within the record.
- The special element name FILLER, which can be used on as many levels and as many times as appropriate, describes an element without naming it. A FILLER element must not be the object of a REDEFINES clause (a FILLER element can, however, redefine another element).

REDefines *base-element-name*

Specifies an alternative description for a previously defined place within the record structure. At runtime, when a program's storage is allocated, the redefining element description will not be allocated new storage space but will, instead, be assigned the same storage as *base-element-name*. *Base-element-name* must be the name of a preceding element of the same level within the record structure. When used, the REDEFINES clause must adhere to the following rules:

- The element containing the REDEFINES clause must not be longer than the base element. Subordinate elements can vary in size, as necessary, within the redefining element or the base element.
- The redefining element cannot be a CALC, sort control, or foreign key element; the base element can be.

- Neither the redefining element nor the base element can be a level-88 description.
- No intervening element (of the same or lower level number) that assigns space can exist between the base element and the redefining element. Other redefining elements, however, are allowed. When an element is redefined more than once, the redefining elements must refer to the name of the base element.
- Neither the redefining element nor its subordinate elements can contain a VALUE clause, except subordinate level-88 elements.
- Elements subordinate to the redefining element can contain REDEFINES clauses.
- Neither the base or redefining element nor their subordinate elements can contain OCCURS DEPENDING ON clauses.
- Elements to which the base and redefining elements are subordinate can contain OCCURS clauses (*without* DEPENDING ON). The base element cannot contain an OCCURS clause, but its subordinate elements can. The redefining element and its subordinate elements can contain OCCURS clauses.

PICture is *picture*

Describes an element by depicting the element's length and data type. PICTURE is not valid for level-88 elements or for elements whose usage is COMPUTATIONAL-1, COMPUTATIONAL-2, or POINTER; For other types of elements, specify *picture* as a 1- to 30-character value that includes only those characters specific to the element's data type. The schema compiler's PICTURE specifications are similar to those for COBOL. See the "Usage" topic for a description of PICTURE specifications for valid data types.

VALue is/VALues are

Assigns an initial value or a list of values to an element description in the application program's main storage at program runtime, or it assigns a conditional value or a list of conditional values to a COBOL condition name (level-88 element). All level-88 element descriptions must include the VALUE clause. Enclose listed values in parentheses.

The VALUE clause has no effect on the database directly; the DBA is encouraged not to include *initial-value* in the data descriptions except as background or null values for use in main storage.

The VALUE clause is prohibited for the following:

- COMP-1, COMP-2, and BIT element descriptions
- An element description containing a REDEFINES clause or an element description subordinate to one containing a REDEFINES clause
- An element description containing an OCCURS clause or an element description subordinate to one containing an OCCURS clause
- An element description of an external floating point number

ALL

Instructs CA IDMS/DB to fill the element description with repetitions of *initial-value*. For example, PIC X(5) VALUE ALL '*' is the same as PIC X(5) VALUE '*****'.

initial-value

Specifies the initial value assigned to the element at runtime as follows:

- **Character string literal**—For alphanumeric elements only: a string of characters enclosed in site-standard quote characters. The character string (including quotes) must not exceed the size of the element as defined in the PICTURE clause or 34 bytes, whichever is shorter.
- **Numeric literal**—For numeric elements only: a string of 1 to 18 numeric characters, optionally preceded by a plus sign (default) or minus sign and optionally containing a decimal point (use the appropriate decimal point character as required by the session option for DECIMAL-POINT).
- **Figurative constant**—For alphanumeric and numeric elements: ZERO, ZEROS, and ZEROES. For alphanumeric elements only: SPACE, SPACES, HIGH-VALUE, HIGH-VALUES, LOW-VALUE, LOW-VALUES, and ALL. ALL is used in conjunction with and indicates repeated occurrences of a nonnumeric literal.

condition-value

Assigns a conditional value to a COBOL condition name (level-88 element). Coding rules specified for *initial-value* above also apply to *condition-value*. *Condition-value* must conform to the picture for the element that occupies storage.

THRu *condition-value*

Specifies a range of valid condition values for COBOL condition names (level 88). When THRU is used, the first *condition-value* assigns the first of a range of values that the condition name will represent at runtime; the second *condition-value* assigns the ending value of the range. To list values or ranges of values, enclose the list in parentheses.

USAge is

Specifies the storage format of data elements. USAGE defaults to CONDITION-NAME for level-88 elements and to DISPLAY for all others.

BIT

Values are stored as bits containing 0s or 1s. Bit elements must always be described in multiples of 8. (CA IDMS/DB does not provide slack bits.) The multiples of 8, however, can range over adjacent elements. For example, five bits can be described in one element and three in the next.

COMPUTATIONAL/COMp/BINary

Numeric values are stored in binary format with the following space requirements:

- 1 to 4 decimal digits require 2 bytes (1 halfword).
- 5 to 9 decimal digits require 4 bytes (1 fullword).
- 10 to 18 decimal digits require 8 bytes (1 doubleword).

COMPUTATIONAL-1/COMP-1/SHOrt-point

Numeric values are stored in internal floating point (short precision) format, requiring 4 bytes. Do not code a PICTURE clause with this usage.

Note: VS2 COBOL does not support COMPUTATIONAL-1.

COMPUTATIONAL-2/COMP-2/LONg-point

Numeric values are stored in internal floating point (long precision) format, requiring 8 bytes. Do not code a PICTURE clause with this usage.

Note: VS2 COBOL does not support COMPUTATIONAL-2.

COMPUTATIONAL-3/COMP-3/PACked

Numeric values are stored in packed decimal format, requiring a half byte for each decimal digit plus a half byte for a sign, rounded up to the next full byte.

CONdition-name

The element does not occupy storage. CONDITION-NAME is assumed if level 88 is specified for the element. Note that CONDITION-NAME can be used in CA ADS dialogs and COBOL programs only. Do not code a PICTURE clause with this usage.

DISplay

Values are stored 1 character to a byte, according to EBCDIC conventions.

DISplay-1

One character occupies 2 bytes. DISPLAY-1 must be specified for double-byte character string (DBCS) data items.

POInter

Values are stored as fullwords. POINTER is used for elements that are to be used as address constants. Do not code a PICTURE clause with this usage.

SYNChronized

Documents the following alignments for usages of COMP, COMP-1, and COMP-2:

- COMP—Halfword (1 to 4 decimal digits) or fullword (5 to 18 decimal digits) alignment
- COMP-1—Fullword alignment
- COMP-2—Doubleword alignment

The SYNCHRONIZED specification does not force alignment, but rather documents user-imposed alignment. If synchronized is specified, filler elements must be used to align numeric data according to the above rules.

OCCurs *occurrence-count* times

Specifies the number of times that the element is to be repeated. *Occurrence-count* must be an unsigned integer in the range 1 through 32,767. Individual occurrences of the element are referenced in application programs by placing a subscript after the element name.

Observe the following rules when using the OCCURS clause:

- An element containing an OCCURS clause cannot be a CALC, sort control, or foreign key element, nor can an element subordinate to an element containing an OCCURS clause be a CALC, sort control, or foreign key element.
- Neither an element containing an OCCURS clause nor an element subordinate to an element containing an OCCURS clause can contain a VALUE clause.
- OCCURS clauses can be nested no more than three deep for use in COBOL programs. Otherwise, any depth of nesting is permissible.

occurrence-count* times DEPending on *control-element-name

Defines a control element within the record that determines the actual number of times the COBOL element will occur.

Occurrence-count must be an integer in the range 1 through 32,767. *Control-element-name* must identify an elementary data element that precedes the element being defined in the record. It must be defined as a signed computational element with a picture in the range S9 through S9(9) or 9 through 9(9). Runtime values of *control-element-name* must be in the range 0 through 32,767 (but not exceeding *occurrence-count*).

Individual OCCURS DEPENDING ON elements are referenced in the same fashion as individual OCCURS elements. Observe the same rules as for the OCCURS clause with the following additions:

- Only one OCCURS DEPENDING ON clause can appear in a record description. The group or elementary item description containing the clause must be the last one in the record description (that is, no element description with the same or lower level number can follow an OCCURS DEPENDING ON element).
- *Control-element-name* cannot contain an OCCURS or REDEFINES clause, nor can it be subordinate to elements that do.
- The element containing an OCCURS DEPENDING ON clause can have subordinate elements that contain OCCURS clauses.

0 to *occurrence-count* times DEPending on *control-element-name*

Indicates that the multiply-occurring group occurs from 0 to *occurrence-count* times depending on the value of the control-element. Rules for *occurrence-count* and *control-element-name* appear above.

JUStify RIGht

Specifies that when the element's runtime value is not as long as the element's picture allows, the value will occupy the rightmost positions of the element. JUSTIFY RIGHT is valid for alphanumeric or alphabetic elements only (group item or one whose PICTURE is specified with Xs or As).

BLAnk when ZERo

Specifies that when the element's runtime value is zero, the value will be changed to spaces.

SIGn is LEAding

Specifies that the sign of a numeric field is to appear in the leading position. This clause is valid for numeric display elements only.

SIGn is TRAiling

Specifies that the sign of a numeric field is to appear in the trailing position. This clause is valid for numeric display elements only.

SEParate character

Causes the sign of a numeric field to appear as a separate byte. This clause is valid for numeric display elements only.

element-synonym-specification

Associates a synonym (alternative name) with the element specified in the ELEMENT substatement. These synonyms are language dependent: each DML precompiler will automatically include the synonym associated with the compiler-specific programming language.

language

Specifies the host language with which the synonym will be used. Valid values are any languages associated with the record's synonyms.

synonym-name

Specifies the name of the synonym to be associated with the primary element name; it must be specified according to the rules for the host language with which the synonym is being used and must follow the rules specified above for element names.

INDexed BY *index-name*

Defines an index to be used at runtime for a multiply-occurring element (that is, one whose definition contains an OCCURS or OCCURS DEPENDING ON clause). This index is used in COBOL SET and SEARCH operations, and is therefore used as a subscript when accessing the associated OCCURS or OCCURS DEPENDING ON element.

Index-name must be a 1- to 30-character name; the characters can be A through Z (at least one), 0 through 9, or the hyphen (except as the first or last character or following another hyphen). It cannot duplicate any element named in the schema. *Index-name* is implicitly defined as a fullword binary item.

You can specify more than one index by creating a list of names enclosed in parentheses.

INDex KEY is *index-name*

Specifies one or more record-specified index keys for a multiply-occurring group record element or a subordinate record element. *Index-name* identifies an elementary element that is subordinate to the associated element. It must be the primary name of the subordinate element; it cannot be a synonym.

You can specify more than one index key by creating a list enclosed in parentheses. Each key can be either ascending or descending.

Note that the INDEX KEY clause allows a mixed collating sequence (that is, a mixture of ascending and descending keys); the ASCENDING/DESCENDING KEY IS clause does not.

ASCending

Sorts the designated key in ascending order.

DEScending

Sorts the designated key in descending order.

ASCending/DEScending KEY is *index-name*

Specifies one or more record-specific index keys for the multiply-occurring group element or subordinate element.

Index-name must be the primary name of an element that is subordinate to the named group element. ASCENDING and DESCENDING sorts the subordinate elements within a multiply-occurring field in ascending or descending order, respectively.

You can specify more than one index key by creating a list enclosed in parentheses.

EDIT TABle is

Specifies an edit table associated with the record element. An edit table contains a list of valid or invalid values for the record element used by the DC/UCF mapping facility.

VALid

Indicates the edit table contains valid values for the record element. VALID is the default.

INValid

Indicates the edit table contains invalid values for the record element.

'*value*'

Specifies a value for the edit table. *Value* is a 1- to 34-character value enclosed in quotes. Separate one value from another with a blank or comma; for example, ('A' 'E' 'G' THRU 'M' 'X').

THRu '*value*'

Specifies a range of values for the edit table.

CODe TABle is

Specifies a translation table to be associated with the record element; for example, a record element containing state abbreviations could have a code table that identifies the name of the state:

```
code table is ('ak' 'alaska' 'al' 'alabama' 'ar' 'arkansas'...)
```

Code tables are used by the DC/UCF mapping facility.

'*encode-value*'

Identifies the value to be translated. *Encode-value* is a 1- to 34-character value enclosed in quotes.

'*decode-value*'

Identifies the translated value. *Decode-value* is a 1- to 64-character value enclosed in quotes. Null values ('') and NOT FOUND are also valid.

EXTernal PICture is *picture*

Defines the display format for record-element data. The picture is available to all map fields that use the record element.

OLQ header

Defines one or more column headers to be used in place of the element name in CA OLQ reports.

CULprit header

Defines one or more column headers to be used in place of the element name in CA Culprit reports.

COMments

Defines comments to be associated with the element description.

DEFinitions

Defines a description of use or purpose for the record element

comment-key

Defines a user-supplied name to be associated with comments about the record element. If *comment-key* contains embedded blanks or delimiters, enclose it in quotes.

comment-text

Specifies text associated with headings, definitions, or comments. *Comment-text* can be any length; nonnumeric literals must be enclosed in quotes. Note, however, that when coding headers, the rules for header definition must be applied to *comment-text*. See the *CA OLQ Reference Guide* or the CA Culprit for further details.

Comment-text can be continued for any number of lines. To continue a header or comment to the next line, code a hyphen in the next line, and code a quote followed by the text of the continued comment after the hyphen. Code a closing quote after the text of the final line.

Comments appear in schema source listings and subschema dictionary listings, and in DML listings when the SCHEMA-COMMENTS option is specified to the DML precompiler.

NULl

Removes text associated with headings, definitions, or comments.

Usage

Naming Elements

When naming schema element types, be sure that the selected names conflict neither with the naming conventions of the programming language(s) that will be used with the CA IDMS Data Manipulation Language (DML) nor with the DML precompilers themselves. As a rule, schema element types should bear names that coincide with the language used most often; use element synonyms to accommodate other languages (see the ELEMENT SYNONYM NAME clause later). In addition to the element naming rules stated above, consider the following points when selecting names (or synonyms) for schema element types:

- **Assembler** names should not exceed eight bytes in length and should not contain hyphens. When the Assembler DML precompiler (IDMSDMLA) generates a DC or DSECT from a schema element description, it uses the element name as the DC or DSECT name. If the element name exceeds eight bytes in length, IDMSDMLA truncates it, possibly causing duplicate names to appear in the program.
- **COBOL** requires names that do not exceed 30 bytes in length and do not contain the characters #, $, or @. When the COBOL DML precompiler (IDMSDMLC) generates a field description from a schema element description, it uses the element name as the field name. If the element name exceeds 30 bytes in length, IDMSDMLC truncates it, possibly causing duplicate names to appear in the program.
- **PL/I** requires names that do not exceed 31 bytes in length and do not contain hyphens. When the PL/I DML precompiler (IDMSDMLP) generates a data field declaration from a schema element description, it changes hyphens in the element name to underscores. If the element name exceeds 31 bytes in length, IDMSDMLP truncates it, possibly causing duplicate names to appear in the program.

SQL synonyms

When using SQL to access a non-SQL defined database, each record in the non-SQL schema is accessed as a table. The name of a column of the table is either:

- The element synonym for language SQL, if one exists
- The element name within the schema record

In either case, hyphens within the name are converted to underscores so that it does not have to be enclosed in quotes within SQL statements.

Elements which occur a fixed number of times within the record have a suffix appended to their name to distinguish occurrences. The suffix is composed of occurrence numbers for each level of nested occurs. For example, if element QUARTERLY-QUOTA occurs 4 times, the corresponding column names are:

- QUARTERLY_QUOTA_1
- QUARTERLY_QUOTA_2
- QUARTERLY_QUOTA_3
- QUARTERLY_QUOTA_4

If QUARTERLY_QUOTA is a sub-element within element ANNUAL- SALES which occurs 3 times, the corresponding column names would be:

- QUARTERLY_QUOTA_1_1...QUARTERLY_QUOTA_1_4
- QUARTERLY_QUOTA_2_1...QUARTERLY_QUOTA_2_4
- QUARTERLY_QUOTA_3_1...QUARTERLY_QUOTA_3_4

Since column names are restricted to 32 characters, it may be necessary to define an SQL synonym for a multiply occurring element so that CA IDMS/DB can append the required suffix.

Function of element level numbers

The function of level numbers 02 through 49 is to create a hierarchy among the element descriptions for a record so that a programmer can, with a single reference, access elements discretely or in groups. The technique is to follow an element description of one level with element description(s) of a higher numbered level. For example, a level 03 element is subordinate to a level 02 element.

Group items and elementary items

A *group item* contains two or more subordinate elements. A DML reference to a group item gains access to all subordinate items. A subordinate item can, in turn, be a group item, with nesting permitted until level 49 is reached (unless otherwise excepted). An item description that has no subordinate items is called an *elementary item*.

The following example outlines the element descriptions for the EMPLOYEE record:

```
02 EMP-ID...                  elementary item
02 EMP-NAME...                group item
   03 EMP-FNAME...            elementary items subordinate
   03 EMP-LNAME...              to EMP-NAME
02 EMP-SEX...                 elementary item
02 EMP-ADDRESS...             group item
   03 EMP-STREET...           elementary items subordinate
   03 EMP-CITY...                  to EMP-ADDRESS
   03 EMP-STATE...
   03 EMP-ZIP...              group item subordinate to
                                    EMP-ADDRESS
      04 EMP-ZIP-FIRST-5...   elementary items subordinate
      04 EMP-ZIP-LAST-4...          to EMP-ZIP
```

Minimum element substatements

The minimum element substatement required for the element to be a *valid* schema component depends on whether the element is a group or elementary item:

- *Group items* require level number and name only.
- *Elementary items* require level number, name, and picture (or usage, where the item's usage prohibits picture specification).

PICTURE formats for alphanumeric data

Alphanumeric data is described by the following characters:

- **X**—The character X represents one alphanumeric character. Note, however, that if USAGE IS BIT (see the USAGE clause in this section), X represents one bit.
- **(n)**—An integer in parentheses can be placed after an X to represent *n* repetitions of the alphanumeric character (for example, X(4) means XXXX).

PICTURE formats for alphabetic data

Alphabetic data is described by the following characters:

- **A**—The character A represents one alphabetic character (A through Z and space only).
- **(n)**—An integer in parentheses can be placed after an A to represent *n* repetitions of the alphabetic character (for example, A(4) means AAAA).

PICTURE formats for DBCS edited data

For DBCS edited data, the PICTURE character string can contain these symbols:

Symbols	Description
G	Each G represents a single DBCS character position (two bytes). When you use this picture, the element USAGE clause must specify DISPLAY-1. Any associated VALUE clause must specify a GRAPHIC literal or the figurative constant SPACES.
B	Each B represents the position used for a space character.

In the following example, the DBCS value represents a string of up to five characters. *So* and *si* represent the shift-out and shift-in characters, respectively:

```
02 zip-code pic g(5)  usage display-1
          value g'sodbcs-valuesi'.
```

PICTURE formats for fixed decimal data

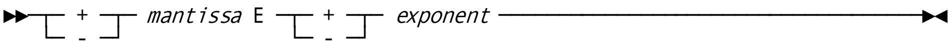
Fixed decimal data is described by the following characters:

- **9**—The character 9 represents one numeric character.
- **(n)**—An integer in parentheses can be placed after a 9 to represent *n* repetitions of the numeric character (for example, 9(4) means 9999).
- **V**—The character V represents an assumed decimal point. No more than one V can appear in an element picture. If the V is omitted and P is not used, the assumed decimal point is after the rightmost 9.
- **P**—The character P represents an assumed zero. Any number of Ps can be placed in the leftmost or rightmost (but not both) positions of an element picture. An assumed decimal point is automatically placed before the first P when the Ps are leftmost and after the last P when the Ps are rightmost.
- **S**—The character S indicates that the number is maintained as either positive or negative. When used, the S must be the first character in the element picture. When the S is omitted, values for the element description are considered positive.

PICTURE formats for external floating point data

External floating point data is described in two parts: the **mantissa**, which represents the decimal part (fractional part) of the element, and the **exponent**, which represents the power of 10 to which the base of one (1) must be raised before being multiplied by the mantissa to determine the element's actual value.

Syntax for the floating point picture is shown next:



Symbol	Description
+/-	The plus sign or the minus sign indicates whether the mantissa is positive or negative.
mantissa	The numeric part of the mantissa is described by the following characters: *9*, which represents one numeric character; *(n)*, following a 9, which represents n repetitions of the numeric character; and V, which represents an assumed decimal point. At least one 9 is required. No more than one V can appear in the mantissa; if the V is omitted, the assumed decimal point is after the rightmost 9.
E	The character E signifies the beginning of the exponential portion of the picture.
+/-	The plus sign or the minus sign indicates whether the exponent is positive or negative.
exponent	The numeric part of the exponent is described by the following characters: *9*, which represents one numeric character; and *(n)*, following a 9, which represents n repetitions of the numeric character. At least one 9 is required; no more than two 9s (or the equivalent 9(2)) can be coded.

PICTURE formats for numeric edited data

Numeric edited data is described by using the numeric data characters described above, along with the following editing characters:

```
Z        +      ,
B        CR     -
0        DB     *
$        .
```

These characters represent edit symbols used in reporting data; quotes are not required. For the individual interpretations of these symbols, refer to the appropriate programming language manual.

Note that if the current decimal point default is DECIMAL-POINT IS COMMA, a period (.) is interpreted as an insertion character and a comma (,) is interpreted as a decimal point.

Data formats described only in elementary items

The actual formats of data can be described only in elementary items. Consequently, the PICTURE, USAGE (except BIT), SYNCHRONIZED, BLANK WHEN ZERO, and SIGN clauses are prohibited in group element descriptions. During programming operations, however, data is accessible not only through its elementary item description, but also through all group items under which it falls. The element EMP-ADDRESS, for example, could be referred to directly in a program.

COBOL condition names

The function of level number 88 is to assign COBOL condition names to specific runtime values of an element. A level 88 element does not occupy storage at runtime: it merely provides a name for a particular value that the preceding element's (level 02 through 49) runtime storage may contain. The name of the level-88 element is known as a condition name. A level-88 ELEMENT substatement must immediately follow either the substatement describing the element for which the level-88 element provides a condition name or another level 88 ELEMENT substatement. The following example illustrates the description of a level-88 element; see the presentation of the VALUE clause for further details:

```
add record name is expertise
    .
    .
    .
    02 skill-level-0425     picture is xx.
       88 expert-0425       usage is condition-name
                            value is '04'.
       88 proficient-0425   usage is condition-name
                            value is '04'.
       88 competent-0425    usage is condition-name
                            value is '04'.
```

```
88 elementary-0425   usage is condition-name
                     value is '04'.
```

Usage clause restrictions for PICTURE clause data types

Alphanumeric, alphabetic, external floating point, and numeric edited descriptions must always have a usage of DISPLAY. Fixed decimal element descriptions can have a usage of DISPLAY, COMP, COMP-3, or COMP-4.

The exact runtime characteristics of an element depend not only on the PICTURE specification, but also on other specifications for the element's format, such as USAGE. The following table illustrates several PICTURE specifications in combination with VALUE specifications.

Usage	Picture	Sample Value	Storage Requirements
DISPLAY	X(5)	T0241	5 bytes
	X(10)	JUNE	10 bytes—Padded on right with blanks
	9(7)	2376600	7 bytes
	9(10)	2376600	10 bytes—Padded on left with zeros
	9(7)V99	2376600.59	9 bytes—Assumed decimal point requires no space
	9(5)PP	2376600	5 bytes—Assumed zeros require no space
	+99E-9	.0000059	6 bytes
DISPLAY-1	G(5)	DBCS character string	10 bytes
COMP	9(4)	2376	2 bytes
	9(7)V99	2376600.59	4 bytes
COMP-1	none	2376600.59	4 bytes
COMP-2	none	2376600.59	8 bytes
COMP-3	9(7)	2376600	4 bytes
	9(7)V99	2376600.59	5 bytes
BIT	X	1	1 byte
	X(7)	FILLER	

How the COBOL DML precompiler handles bit elements

When a COBOL program copies a record that contains a bit element, the DML precompiler does the following:

- If the bit element starts on a byte boundary, it assigns a usage of DISPLAY and a picture of X(*n*); *n* is the number of bytes before the next bit item that starts on a byte boundary.
- If the bit element does not start on a byte boundary, it is not reflected in the COBOL program.

Element storage characteristics due to usage and picture

The following table illustrates how values are stored with different usages.

Usage	Alphanumeric Value	Internal Representation in hexadecimal
DISPLAY	BILL BALL	C2 C9 D3 D3 40 C2 C1 D3 D3
DISPLAY	4857964	F4 F8 F5 F7 F9 F6 F4
COMP	4857964	00 4A 20 6C
COMP-1	4857964	40 4A 20 6C
COMP-2	4857964	40 00 00 00 00 4A 20 6C
COMP-3	4857964	48 57 96 4C
BIT	B'11110000'	F0
POINTER	4857964	00 4A 20 6C

OCCURS DEPENDING ON creates variable-length records

The OCCURS DEPENDING ON clause makes a record variable in length. If the MINIMUM ROOT LENGTH and/or MINIMUM FRAGMENT LENGTH clauses are not included in the record description, the defaults (CONTROL LENGTH and four bytes) are assigned. The total space required in main storage for a variable-length record is:

```
main storage space = F + (V * M)

    where F = the length of the record's fixed
              portion
          V = the length of one occurrence of
              the record's variable portion
          M = the maximum number of times the control element
              can occur
```

For example, the total main storage required for the ABRIDGED-DENTAL-CLAIM record described next under "Examples" is 20 + 15 + 2 + 9 + 2 + ((2 + 2 + 2 + 2) * 10) = 128 bytes. The actual size of a specific occurrence of the record (data portion) as stored in the database, however, is as follows:

```
database storage space = F + (V * C)

    where F = the length of the record's fixed
              portion
          V = the length of one occurrence of
              the record's variable portion
          C = the value of the control element
              in the specific record occurrence.
```

A value of 2 for DC-NUMBER-OF-PROCEDURES, for example, indicates two DC-DENTIST elements and a record length of 20 + 15 + 2 + 9 + 2 + ((2 + 2 + 2 + 2) * 2) = 64 bytes.

SQL Considerations

If you intend to use SQL to access the data described by a non-SQL schema record, consider the following when designing your record elements:

- Group elements are not visible as columns in SQL, but elementary items within group elements are
- Fillers, condition names, redefining elements and elements subordinate to redefining elements are not visible as columns
- Elements containing an OCCURS DEPENDING ON clause and elements subordinate to such an element are not visible as columns
- The datatype of a column is derived from the picture and usage of the corresponding element as follows:

Picture and Usage	Data Type
PIC X(n) usage DISPLAY	CHAR(n)
PIC A(n) usage DISPLAY	CHAR(n)
Numeric edited1	CHAR(l), l=byte length
External floating point2	CHAR(l), l=byte length
PIC G(n) usage DISPLAY	GRAPHIC(n)
PIC S9(t)V9(s) usage DISPLAY	NUMERIC(t+s,s)
PIC SP..9(p) usage DISPLAY3	NUMERIC(p,p)
PIC S9(p)P.. usage DISPLAY3	NUMERIC(p,0)
PIC 9(t)V9(s) usage DISPLAY	UNSIGNED NUMERIC(t+s,s)

Picture and Usage	Data Type
PIC P..9(p) usage DISPLAY3	UNSIGNED NUMERIC(p,p)
PIC 9(p)P.. usage DISPLAY3	UNSIGNED NUMERIC(p,0)
PIC S9(t)V9(s) usage COMP-3	DECIMAL(t+s,s)
PIC SP..9(p) usage COMP-33	DECIMAL(p,p)
PIC S9(p)P.. usage COMP-33	DECIMAL(p,0)
PIC 9(t)V9(s) usage COMP-3	UNSIGNED DECIMAL(t+s,s)
PIC P..9(p) usage COMP-33	UNSIGNED DECIMAL(p,p)
PIC 9(p)P.. usage COMP-33	UNSIGNED DECIMAL(p,0)
PIC S9(n), n<5 usage COMP4	SMALLINT
PIC S9(n), 4<n<10 usage COMP4	INTEGER
PIC S9(n), 9<n usage COMP4	LONGINT
PIC 9(n) usage COMP4	BINARY(l), l=byte length
PIC X(n) usage BIT	BINARY(l), l=byte length
USAGE POINTER	BINARY(4)
USAGE COMP-1	REAL
USAGE COMP-2	DOUBLE PRECISION

1. Numeric edited includes any element whose usage is DISPLAY and:

- Whose picture contains any of the editing symbols: + - Z B 0 $ CR DB . , *
- Whose picture clause contains only the symbols: 9 (n) V S P but whose element description also includes the SIGN LEADING or SEPARATE CHARACTER specification

2. External floating point includes any element whose usage is DISPLAY and whose picture is: +/- mantissa E +/- exponent

3. The scaling character "P" in a picture clause is ignored in value representations of associated columns. This has the effect of representing values of such columns as a power of 10 greater than or smaller than their actual value. For example, if an element is described as PIC S9(5)PPP, a value of 123000 will be represented in SQL as 123. If an element is described as PIC SPPP9(5), a value of .000123 will be represented in SQL as .123.

4. Computational elements also include those whose USAGE is BINARY and COMP-4. If the picture of a computational item includes an implied decimal point, it is ignored in determining the data type of the column. This has the effect of representing values of such columns as a power of 10 greater than their actual values. For example, if an element is described as PIC S9(5)V99 USAGE COMP, a value of 123.56 will be represented in SQL as 12345.

Elements whose usage is BIT are not represented by columns except as noted:

- Group elements in which all subordinate elements have a usage of BIT and which start on a byte boundary are represented by columns with a data type of BINARY. The length of the column is the length in bytes from the start of the group element to the start of the next element at the same level which begins on a byte boundary. If groups are nested within groups, the group element with the lowest level number in which all subordinate elements are bits is the element represented by a column. Intervening and subordinate elements are not represented by columns.
- BIT elements occurring a fixed number of times and beginning on a byte boundary are represented by columns with a data type of BINARY. The length of the column is the length in bytes from the start of the element to the start of the next element at the same level which also begins on a byte boundary. Intervening elements are not represented by columns.
- Other BIT elements which begin on a byte boundary are represented by columns with a data type of BINARY. The length of the column is the length in bytes from the start of the element to the start of the next element at the same level which also begins on a byte boundary. Intervening elements are not represented by columns.

Examples

Minimum element substatement

Minimal ELEMENT substatements are illustrated next:

```
02  claim-date.
    03  claim-year   picture 99.
    03  claim-month  picture 99.
    03  claim-day    picture 99.
```

A valid element description also requires usage information. In the above example, the schema compiler defaults to assign USAGE IS DISPLAY to each element.

Redefining the same element storage area

In the following example, one record type holds data relating to four different types of facilities and, accordingly, requires four definitions of the same storage area:

```
modify record name is facility.
   02 fc-id                     pic x(4).
   02 fc-lunchroom.
      03 fc-l1-length           pic 99.
      03 fc-l1-width            pic 99.
      03 fc-l1-tables           pic 99.
      03 fc-l1-seats            pic 9(4).
      03 fc-l1-pots             pic 99.
   02 fc-lounge                 redefines fc-lunchroom.
      03 fc-l2-chairs           pic 99.
      03 fc-l2-ashtrays         pic 99.
      03 fc-l2-tables           pic 99.
      03 filler                 pic 9(6).
   02 fc-emp-library            redefines fc-lunchroom.
      03 fc-l3-desks            pic 99.
      03 fc-l3-tables           pic 99.
      03 fc-l3-bookcases        pic 99.
      03 fc-l3-mag-racks        pic 99.
      03 filler                 pic 9(4).
   02 fc-hallway                redefines fc-lunchroom.
      03 fc-h-length            pic 99.
      03 fc-h-width             pic 99.
      03 filler                 pic 9(8).
```

Base-element-name cannot contain OCCURS clause

In the following example, any element *except EXP-SKILL-DATE-N* can contain an OCCURS clause:

```
05  exp-skill-date.
    10  exp-skill-date-n.
        15  exp-skill-year-n    pic 99.
        15  exp-skill-month-n   pic 99.
        15  exp-skill-day-n     pic 99.
    10  exp-skill-date-x  redefines exp-skill-date-n.
        15  exp-skill-year-x    pic 99.
        15  exp-skill-month-x   pic 99.
        15  exp-skill-day-x     pic 99.
```

Group elements have implied pictures

In this example, group elements, COV-SELECT-DATE and COV-TERMIN-DATE have implied pictures of X(6). Group elements have implied pictures of X(*n*), where *n* equals the total number of bytes required by all subordinate elements.

```
modify record name is coverage.
   02 cov-select-date.
      02 cov-select-year      pic 99.
      02 cov-select-month     pic 99.
      02 cov-select-day       pic 99.
   02 cov-termin-date.
      02 cov-termin-year      pic 99.
      02 cov-termin-month     pic 99.
      02 cov-termin-day       pic 99.
   02 cov-type                pic x.
   02 cov-insplan-code        pic xxx.
```

Assigning condition values to level-88 elements

These two examples show different ways of assigning condition values for the same record definition:

- Example 1:

```
modify record name is structure.
   02 struct-code                 pic xx.
      88 president                value 'a1'.
      88 sr-vice-president        value 'a2'.
      88 vice-president           value 'a3'.
      88 sr-manager               value 'b1'.
      88 mid-manager              value 'b2'.
      88 lower-manager            value 'b3'.
      88 supervisor               value 'c1'.
      88 senior                   value 'd1'.
      88 regular                  value 'd2'.
      88 trainee                  value 'd3'.
   02 struct-effective-date.
      03 struct-effect-year       pic 99.
      03 struct-effect-month      pic 99.
      03 struct-effect-day        pic 99.
```

- Example 2:

```
modify record name is structure.
   02 struct-code                 pic xx.
      88 president                value 'a1'.
      88 vice-presidents          value ('a2' 'a3').
      88 managers                 value 'b1' thru 'b3'.
      88 supervisor               value 'c1'.
      88 technicians              value ('d1' 'd2' 'd3').
   02 struct-effective-date.
      03 struct-effect-year       pic 99.
      03 struct-effect-month      pic 99.
      03 struct-effect-day        pic 99.
```

In a COBOL program using this record description, the following statements have the same meaning:

```
if president then perform 0500-bigwig.

if struct-code = 'a1' then perform 0500-bigwig.
```

Variable-length record description

The following example describes a variable number of DC-DENTIST-CHARGES elements within the ABRIDGED-DENTAL-CLAIM record type:

```
modify record name is abridged-dental-claim.
   02  dc-dentist-address.
       03  dc-dent-street        pic x(20).
       03  dc-dent-city          pic x(15).
       03  dc-dent-state         pic xx.
       03  dc-dent-zip           pic x(9).
   02  dc-number-of-procedures   pic 99  comp.
   02  dc-dentist-charges      occurs 0 to 10 times
                               depending on
                               dc-number-of-procedures.
       03  dc-tooth-number       pic 99.
       03  dc-service-date.
           03  dc-serv-year      pic 99.
           03  dc-serv-month     pic 99.
           03  dc-serv-day       pic 99.
```

Repeating group items

The following example defines eight occurrences of the DC-CLAIM-DATE element:

```
02 dc-claim-date            occurs 8 times.
   03 dc-claim-year         pic 99.
   03 dc-claim-month        pic 99.
   03 dc-claim-day          pic 99.
```

The total length of all DC-CLAIM-DATE elements is 8 * (2 + 2 + 2) = 48 bytes. To reference the second DC-CLAIM-DATE element, the programmer can code DC-CLAIM-DATE(2) or DC-CLAIM- DATE(*subscript*), where *subscript* is an elementary item that contains the value 2. To reference only the DC-CLAIM-MONTH element of the second DC-CLAIM-DATE element, the programmer can code DC-CLAIM-MONTH(2) or DC-CLAIM-MONTH(*subscript*).

The previous example can be expanded as follows to include a second level of multiply-occurring elements:

```
02 dc-claim-date            occurs 8 times.
   03 dc-claim-year         pic 99.
   03 dc-claim-month        pic 99.
   03 dc-claim-day          pic 99.
   03 dc-claim-time         occurs 6 times.
      05 dc-claim-hour      pic 9.
      05 dc-claim-am-or-pm  pic xxxx.
```

The total length of the DC-CLAIM-DATE element now is 8 * ((2 + 2 + 2) + (6 * (1 + 4))) = 288 bytes. To refer to the fourth DC-CLAIM-TIME element subordinate to the second DC-CLAIM-DATE element, the programmer can code DC-CLAIM-TIME(2,4) or DC-CLAIM-TIME(*subscript-1, subscript-2*), where *subscript-1* is an elementary item that contains the value 2 and *subscript-2* is an elementary item that contains the value 4.

Indexing a multiply-occurring element

In the following example, the DC-DENTIST-CHARGES element defines an index named DCX:

```
02  dentist-charges-0405
    occurs 0 to 10 times
    depending on number-of-procedures-0405
    indexed by dcx.
```

Associating comments with element descriptions

The following example illustrates the use of element comments in the COVERAGE record:

```
modify record name is coverage.
   02 cov-select-date.
      02 cov-select-year      pic 99.
      02 cov-select-month     pic 99.
      02 cov-select-day       pic 99.
   02 cov-termin-date.
      02 cov-termin-year      pic 99.
      02 cov-termin-month     pic 99.
      02 cov-termin-day       pic 99.
   02 cov-type                pic x.
      comments 'this is the type assigned to the coverage by
      -        'our company''s insurance professionals'.
   02 cov-insplan-code        pic xxx.
      comments 'this is the code assigned to the coverage by
      -        'the insurance company'.
```

More Information

- For more information about mixing element substatements with the COPY ELEMENTS substatements, see "Usage" under COPY ELEMENTS.
- For more information about code tables and external pictures, see the *CA IDMS Mapping Facility Guide*.

COPY ELEMENTS Substatement

The COPY ELEMENTS substatement requests inclusion of all elements from a record description already stored in the dictionary. The record description may have been stored through another schema or the IDD DDDL compiler. COPY ELEMENTS can be used in place of ELEMENT substatements to define all of the record's elements or only some of them. When COPY ELEMENTS supplies some of the record's elements, use ELEMENT substatements to supply the rest.

Unlike the SHARE clause of the RECORD statement, COPY ELEMENTS generates a new copy of the record structure for *record-name* (the object of the ADD or MODIFY).

Syntax

COPY ELEMENTS substatement

```
►►─── COPy ELements from record base-record-name ─────────────────────────────►

►─┬───────────────────────────────────────────────────────────────┬──────────►◄
  ├─ version-specification ───────────────────────────────────────┤
  └─ of SCHema base-schema-name ─┬─────────────────────────────┬──┘
                                 └─ version-specification ─────┘
```

Parameters

COPy ELements from record *base-record-name*

Identifies the record whose structure is to be copied into the description of *record-name* (the object of the ADD or MODIFY). Copied elements have the same level numbers in *record-name* that they have in the base record.

Base-record-name must identify a record already defined in the dictionary and can be a primary name or a synonym (as described under "RECORD statements," in this chapter).

version-specification

Uniquely qualifies *base-record-name* with a version number. The default is the current session option for existing versions.

Note: Expanded syntax for *version-specification* is presented in Chapter 13, “Parameter Expansions".

of SCHema *base-schema-name*

Qualifies descriptions of records that participate in a schema. *Base-schema-name* must be the name of a schema, already defined in the dictionary, in which *base-record-name* participates.

version-specification

Uniquely qualifies *base-schema-name* with a version number. The default is the current session option for existing versions.

Note: Expanded syntax for *version-specification* is presented in Chapter 13, “Parameter Expansions".

Usage

Mixing Element and COPY ELEMENTS Substatements

Element and COPY ELEMENTS substatements can be mixed in any sequence necessary to describe the structure of the record. However, because the level numbers of copied elements are the same as those in the base record, you should exercise care in mixing elements of different levels. To mix element and COPY ELEMENTS substatements and to change the level numbers within the record, do the following:

1. Code ELEMENT and COPY ELEMENTS substatements to place the elements into their appropriate positions, as shown in the example that follows this discussion.
2. Online, issue a DISPLAY RECORD with AS SYNTAX and VERB MODIFY for the record; in batch mode, code PUNCH instead of DISPLAY.
3. Change the affected level numbers only. *Do not erase unaffected elements*: all elements for a single record must always be presented together.
4. Submit the new statement to the compiler.

Examples

In the following example, the structure of NEW-COVERAGE is generated by copying elements from the COVERAGE record and the DDDL-built CARRIER-DETAIL record, and by coding new element descriptions in line.

```
add record name is new-coverage
    location mode is via emp-coverage set
    within emp-demo-region area.
    copy elements from record coverage
        of schema empschm version 1.
    02  cov-carrier-id          pic 99.
    02  cov-carrier-name        pic x(20).
    copy elements from record carrier-detail.
```

The previous example effectively produces a new record description, NEW-COVERAGE, that has the following structure:

```
01 new-coverage.
   02 cov-select-date.
      03 cov-select-year   pic 99.
      03 cov-select-month  pic 99.
      03 cov-select-day    pic 99.
   02 cov-termin-date.
      03 cov-termin-year   pic 99.
      03 cov-termin-month  pic 99.
      03 cov-termin-day    pic 99.
   02 cov-type             pic x.
   02 cov-insplan-code     pic xxx.
   02 cov-carrier-id       pic 99.
   02 cov-carrier-name     pic x(20).
   02 cov-carr-no-of-claims
                           pic 99 comp.
   02 cov-carr-claims-processed
                 occurs 0 to 100
                 depending on
                 cov-carr-no-of-claims.
      03 cov-carr-payment  pic x.
        88 prompt          value '9'.
        88 over-30-days    value '4'.
        88 over-60-days    value '1'.
      03 cov-carr-courtesy pic x.
      03 cov-carr-check    pic x.
         88 cleared        value 'c'.
         88 bounced        value 'b'.
```

SET Statement

The SET statements identify and describe a set. Depending on the verb, the SET statements can add, modify, delete, display, or punch the set description.

The schema compiler applies SET statements to the current schema.

Note: For an explanation of schema currency see 9.7, “Establishing Schema and Subschema Currency”.

Syntax

ADD/MODIFY SET statement

```
►►─┬─ ADD ────┬─ SET name is set-name ──────────────────────────────────────►
   └─ MODify ─┘

 ►─┬────────────────────────────────────────────────────────────────────────►─
   └─ SAMe AS SET base-set-name ────────────────────────────────────────────

─►──────────────────────────────────────────────────────────────────────────►
  ─── of SCHema base-schema-name ─┬──────────────────────────────┬┘
                                  └─ version-specification ─┘

 ►─┬──────────────────────────────────────┬─────────────────────────────────►
   └─ ORDer is ──┬─ FIRst ────────────────┘
                 ├─ LASt ───────────┤
                 ├─ NEXt ───────────┤
                 ├─ PRIor ──────────┤
                 └─ SORted ─────────┘

 ►─┬──────────────────────────────────────────────────────────────┬─────────►
   └─ MODe is ──┬─ CHAin ─┬────────────────────────────────────┬──┘
                │         └─ LINked to PRIor ─┘                │
                ├─ VSAm INDex ─────────────────────────────────┤
                └─ INDex indexed-set-mode-specification ───────┘

 ►─┬─────────────────────────────────────────────────────────────┬──────────►
   ├─ OWNer is record-name ─┬──────────────────────────────┬─────┤
   │                        └─ owner-record-options ─┘         │
   └─ OWNer is SYStem ──┬─────────────────────────────┬────────┘
                        └─ area-specification ─┘

 ►─┬──────────────────────────────────────────────────────────────────┬──►◄
   │ ┌────────────────────────────────────────────────────────────┐   │
   └─▼─┬─────────────────┬─ MEMber is record-name ─┬─────────────────┬┴───┘
       ├─ INClude ◄──────┤                         └─ member-record-options ─┘
       └─ EXClude ───────┘
```

Expansion of *indexed-set-mode-specifications*

```
►►─┬─ USIng symbolic-index-name ───────────────────────────────────────┬─►◄
   └─ BLOck CONtains key-count keys ─┬─────────────────────────────────┤
                                     └─ DISplacement is ─┬─ 0 ◄────────┘
                                                         └─ page-count ─┘
```

Expansion of *owner-record-options*

```
►►─┬────────────────────────────────────────────────────────┬───────────►
   └─ NEXt dbkey POSition is ──┬─ next-dbkey-position ─┬────┘
                               └─ AUTo ────────────────┘

 ►─┬────────────────────────────────────────────────────────┬───────────►
   └─ PRIor dbkey POSition is ─┬─ prior-dbkey-position ─┬───┘
                               └─ AUTo ─────────────────┘

 ►─┬──────────────────────────────────────────────────────┬───────────►◄
   └─ PRImary KEY is ──┬─ system-owned-index-name ─┬───────┘
                       ├─ CALc ────────────────────┤
                       └─ NULl ────────────────────┘
```

Expansion of ***area-specification***

```
►►─── WIThin AREa area-name ─────────────────────────────────────────────►

 ►─┬──────────────────────────────────────────────────────────────────────┬─►◄
   ├─ SUBarea symbolic-subarea-name ──────────────────────────────────────┤
   └─ OFFset ─┬─ 0 ◄──────────────────────── for ─┬─ 100 PERcent ◄──────┤
              ├─ offset-page-count PAGes ─┤         ├─ percent PERcent ───┤
              └─ offset-percent PERcent ──┘         └─ page-count PAGes ──┘
```

Expansion of ***member-record-options***

```
►►─┬──────────────────────────────────────────────────────────┬──────────►
   └─ INDex dbkey POSition is ─┬─ OMItted ───────────────────┤
                               ├─ index-dbkey-position ──────┤
                               └─ AUTo ──────────────────────┘

 ►─┬───────────────────────────────────────────────────────┬─────────────►
   └─ NEXt dbkey POSition is ─┬─ next-dbkey-position ─────┤
                              └─ AUTo ────────────────────┘

 ►─┬────────────────────────────────────────────────────────┬────────────►
   └─ PRIor dbkey POSition is ─┬─ prior-dbkey-position ────┤
                               └─ AUTo ────────────────────┘

 ►─┬───────────────────────────────────────────────────────────────────┬─►
   └─ LINked to OWNer ─┬──────────────────────────────────────────────┤
                       └─ OWNer dbkey POSition is ┬ owner-dbkey-position ┤
                                                  └ AUTo ────────────────┘

 ►─┬───────────────────────────────────────────────────────────────────┬─►
   └─ FOReign KEY is ─┬──────────────────────────────────────────────┤
                      ├─ element-name ─────┬──────────────┬─────────┤
                      │                    └─ NULlable ───┘         │
                      │                                             │
                      │      ┌───────────────────────────────┐      │
                      │      ▼                               │      │
                      ├─ ( ── element-name ──┬──────────────┬┴─ ) ──┤
                      │                      └─ NULlable ───┘       │
                      └─ NULl ──────────────────────────────────────┘

 ►─┬─ MANdatory ─┬──┬─ AUTomatic ─┬──────────────────────────────────────►
   └─ OPTional ──┘  └─ MANual ────┘

 ►─┬───────────────────┬──────────────────────────────────────────────────►◄
   └─ key-expression ──┘
```

Expansion of ***key-expression***

```
►►─┬──────────────────┬─ KEY is ───────────────────────────────────────────►
   ├─ ASCending ─────┤
   └─ DEScending ────┘

 ►─┬─ sort-element-name ─┬─────────────────┬───────────────────┬─────────►
   │                     ├─ ASCENDING ◄────┤                   │
   │                     └─ DEScending ────┘                   │
   │                                                           │
   │      ┌──────────────────────────────────────────────┐     │
   ├─ ( ──▼─ ( sort-element-name ─┬─────────────────┬────┴─ ) ─┤
   │                              ├─ ASCending ◄────┤          │
   │                              └─ DEScending ────┘          │
   └─ DBKey ─┬─────────────────┬──────────────────────────────┘
             ├─ ASCending ◄────┤
             └─ DEScending ────┘
```

```
►─┬────────────────────┬─┬──────────────────┬────────────────────────────────►
  └─ NATural sequence ─┘ ├─ COMpressed ─────┤
                         └─ UNCOMpressed ───┘

►─┬───────────────────────────────────────────┬──────────────────────────────►◄
  └─ DUPlicates are ─┬─ FIRst ──────────────────┘
                     ├─ LASt ─────────────┤
                     ├─ UNORDered ────────┤
                     ├─ NOT allowed ──────┤
                     └─ by DBKey ─────────┘
```

DELETE SET statement

```
►►── DELete SET name is set-name ──────────────────────────────────────────────►◄
```

DISPLAY/PUNCH SET statement

```
►►─┬─ DISplay ─┬─ SET name is set-name ────────────────────────────────────────►
   └─ PUNch ───┘

►─┬─────────────────────────────────────────────────┬──────────────────────────►
  │  ┌─────────────────────────────────────────────┐ │
  │  │                      ┌──────────────────┐  │ │
  └─▼─┬─ WITh ─────────┬─▼─┬─ DETails ──────┴──┴─┘
      ├─ ALSo WITh ────┤   ├─ ALL ─────────┤
      └─ WITHOut ──────┘   └─ NONe ────────┘

►─┬────────────────────────┬───────────────────────────────────────────────────►
  └─ VERB ─┬─ ADD ──────────┘
           ├─ MODify ────┤
           ├─ DELete ────┤
           ├─ DISplay ───┤
           └─ PUNch ─────┘

►─┬────────────────────────┬───────────────────────────────────────────────────►
  └─ AS ─┬─ COMments ──────┘
         └─ SYNtax ─────┘

►─┬─────────────────────────────────────┬──────────────────────────────────────►◄
  └─ TO ─┬─ module-specification ───────┘
         └─ SYSpch ─────────────────┘
```

Parameters

SET name is *set-name*

Identifies the database set description. *Set-name* must be a 1- to 16- character name. Apply the following considerations when selecting set names:

- *Set-name* must not be the same as the schema name or the name of any other component (including synonyms) within the schema.
- Because *set-name* will be copied into DML programs, it must not be the name of a keyword known to either the DML precompiler or the host programming language.

SAMe AS SET *base-set-name*

Copies the entire set description (order, mode, owner, and members) from *base-set-name* of another schema into the description *set-name* (the object of the ADD or MODIFY). *Base-set-name* must identify an existing set.

of SCHema *base-schema-name*

Identifies the schema that contains *base-set-name*. The base schema must have a status of VALID.

version-specification

Uniquely qualifies the schema with a version number. The default is the current session option for existing versions. If the schema version that corresponds to HIGHEST or LOWEST does not contain *base-set-name*, the schema compiler issues an error message.

Note: Expanded syntax for *version-specification* is presented in the chapter “Parameter Expansions".

ORDer is

Specifies the logical order of adding new member record occurrences to a set occurrence at runtime.

FIRst

Positions the new record immediately after the owner record, becoming the first member in the set (a LIFO stack).

LASt

Positions the new record immediately before the owner record, becoming the last member in the set (a FIFO stack). If MODE IS CHAIN is also coded, include LINKED TO PRIOR in the MODE clause.

NEXt

Positions the new record immediately after the current of set.

PRIor

Positions the new record immediately before the current of set. If MODE IS CHAIN is also coded, include LINKED TO PRIOR in the MODE clause.

SORted

Positions the new record according to the value of one or more of its data elements (called a sort control element) relative to the values of the same elements in other member records of the same type. ORDER IS SORTED must be specified for native VSAM sets.

MODe is

Specifies the characteristic of the set that tells CA IDMS/DB how pointers are to be maintained at runtime.

CHAin

Links each record in the set to the next record (establishes the NEXT pointer for the set) and is mandatory for all set types except indexed sets and native VSAM sets.

LINked to PRIor

Specifies that each record in a chained set will be chained to the prior record (establishes the PRIOR pointer for the set) as well as to the next record. LINKED TO PRIOR is required if LAST or PRIOR was specified in the ORDER clause.

When using LINKED TO PRIOR and assigning pointers manually (see the OWNER and MEMBER clauses, later), be sure to code the PRIOR DBKEY POSITION clause of the OWNER and MEMBER clauses.

VSAm INDex

Identifies the set as a native VSAM set representing either a primary index on a KSDS file or an alternate index on an ESDS or KSDS file. Each VSAM set must be represented by a KSDS or PATH file in the physical database definition.

VSAM sets can have, as members, only records whose location mode is VSAM OR VSAM CALC; owner records are not specified for VSAM sets.

INDex *indexed-set-mode-options*

Identifies the set as an indexed set. This option is not valid for multiple-member sets.

USIng *symbolic-index-name*

Specifies the name of a symbol representing the index. The symbolic index is assigned values in a corresponding physical area definition that identify either:

- The number of entries in each bottom-level index (SR8) record and, optionally, the displacement of the bottom-level index records from their owners
- The values required by CA IDMS/DB to calculate the number of entries in each bottom-level (SR8) record and its displacement from its owner

INDex BLOck contains *key-cnt* keys

Establishes the number of entries in each bottom-level index record (SR8 system record). *Key-cnt* must be an unsigned integer in the range 3 through 8180.

Note: For the rationale used in determining a value for *key-cnt*, see the *Database Design Guide*.

DISplacement is *page-cnt* pages

Indicates how far away from their owners the bottom level index records are to be stored. *Page-cnt* must be an unsigned integer in the range 0 through 32,767; 0 is the default.

OWNer is *record-name*

Identifies the record type that owns the set; *record-name* must name a record associated with the current schema. This format of the OWNER clause is required for:

- Chained sets
- Indexed sets in which the owner is a user-defined record (see also the OWNER IS SYSTEM clause)

It is not allowed for native VSAM sets.

owner-record-options

Identifies the positions within the owner record's prefix to be used for next and prior (if any) pointers of the set being described and optionally identifies the owner record's primary key.

The defaults for next and prior pointer positions depend on the set's mode as shown in the table under the "Usage" topic. The defaults for each set mode are:

- MODE IS CHAIN causes a default of NEXT DBKEY POSITION IS AUTO; the LINKED TO PRIOR clause causes a default of PRIOR DBKEY POSITION IS AUTO.
- MODE IS VSAM is not applicable to next and prior set pointers.
- MODE IS INDEX causes defaults of NEXT DBKEY POSITION IS AUTO and PRIOR DBKEY POSITION IS AUTO, unless OWNER IS SYSTEM is also coded.

next-dbkey-position

Represents the sequential position of the NEXT set pointer within the owner record's prefix; it must be a whole integer in the range 1 through 8180.

prior-dbkey-position

Represents the sequential position of the PRIOR set pointer within the owner record's prefix; it must be a whole integer in the range 1 through 8180.

When assigning pointer positions manually, remember to specify a prior db-key position if either of these conditions is true:

- LINKED TO PRIOR is specified in the MODE clause.
- INDEX is specified in the MODE clause and OWNER IS SYSTEM is not specified.

AUTo

Causes the schema compiler to automatically assign a set pointer position within the owner record's prefix when the schema description is validated. Until the schema description is validated, a DISPLAY or PUNCH of the set will indicate AUTO for pointer positions; after the schema description has been validated, DISPLAY or PUNCH indicates the sequential pointer positions that the validation resolved (see 14.8, “VALIDATE Statement").

PRImary KEY is

For SQL access against a non-SQL defined database, defines a primary key field in the owner record.

system-owned-index-name

Identifies a system-owned index as the primary key. To use this specification, the owner record must be a member of the named index and the named index must be a mandatory automatic set defined as duplicates not allowed. No elements named as the keys for the system-owned index can be group elements.

CALc

Identifies the primary key as the owner record's CALC key. To use this specification, the owner record must be stored with a location mode of CALC in which duplicates are not allowed. The CALC key must not contain a group element.

NULl

Removes the primary key from the set and all foreign keys associated with the primary key.

OWNer is SYStem

Specifies that the indexed set being described is owned by an internal owner record (SR7 system record). A single occurrence of the SR7 record type owns the set containing all member occurrences (identified in the MEMBER clause, shown next). OWNER IS SYSTEM establishes a relationship that is functionally, though not internally, the same as that of a one-of-a-kind (OOAK) record to its set members.

OWNER IS SYSTEM is not valid in the following instances:

- If the set mode is CHAIN
- If the set mode is VSAM INDEX

area-specification

Specifies the area in which the owner record (SR7) and the index structure is to reside. If this clause is not coded, the owner record and index structure will be stored in the same area as the member record (specified in the MEMBER clause).

WIThin AREa *area-name*

Specifies the name of the area. *Area-name* must be the name of an area already defined as part of the current schema.

Defaults for the WITHIN AREA clause are as follows:

- If WITHIN AREA is coded with neither SUBAREA nor OFFSET, the SR7 owner record is stored within the named area's page range.
- If WITHIN AREA is not coded, CA IDMS/DB will place the owner record in the same area and page range as the set member (in the MEMBER clause).

SUBarea *symbolic-subarea-name*

Names a symbol representing a page range (or subarea). Within the physical area definition, the symbolic subarea is assigned the actual range of pages in which CA IDMS/DB will store the system-owned index structure.

OFFset

Specifies a relative range of pages in the physical area, in terms of either a percentage of the area or a number of pages, in which CA IDMS/DB will store the owner record and the index structure.

***offset-page-count* PAGes**

Determines the first page in which CA IDMS/DB will store the owner record based on the lowest page number of the area:

```
record lopage = (LPN + offset-page-count)

     where LPN = the lowest page number in the physical area
```

Offset-page-count must be an integer in the range 0 through the number of pages in *physical-area-name* minus 1.

***offset-percent* PERcent**

Determines the first page in which CA IDMS/DB will store the owner record based on the initial page range of the physical area:

```
record's lopage = (LPN + (INP * offset-percent * .01))

     where LPN = the lowest page number in the physical area
       and INP = the initial number of pages in the physical area
```

Offset-percent must be an integer in the range 0 through 100.

FOR *page-count* PAGes

Determines the last page in which CA IDMS/DB will store the owner record based on the record's low page:

```
record's hipage = (RLP + page-count - 1)

     where RLP = the first page in which the SR7 can be stored
```

The calculated page must not exceed the highest page number in the physical area.

FOR *percent* PERcent

Determines the last page in which CA IDMS/DB will store the owner record based on the record's low page and the total number of pages in the physical area:

```
record's hipage = (RLP + (TNP * percent * .01) - 1)

     where RLP = the first page in which the SR7 can be stored
       and TNP = the total number of pages in the physical area
```

Percent must be an integer in the range 1 through 100. The default is 100. If *percent* causes the calculated high page to be greater than the highest page number in the physical area, CA IDMS/DB will ignore the excessive page numbers, and will store the record occurrences up to and including the last page in the physical area.

INClude MEMber is *record-name*

Identifies a record type that is to participate as a member of the set. *Record-name* must name a record associated with the current schema. Code as many MEMBER clauses as are necessary to declare all of the set's member record types (note that indexed sets and native VSAM sets must include only one member record type).

EXClude MEMber is *record-name*

Identifies a record type that is no longer to participate as a member of the set. *Record-name* must name a record type that was previously included in the set definition. Additional options of the MEMBER clause are invalid.

member-record-options

Specifies additional information about set members in order to maintain the set at runtime.

AUTo

Causes the schema compiler to automatically assign a set pointer position within the member record's prefix when the schema description is validated. Until the schema description is validated, a DISPLAY or PUNCH of the set will indicate AUTO for pointer positions; after the schema description is validated, DISPLAY or PUNCH indicates the pointer positions that the validation resolved.

Defaults assigned by the schema compiler depend on the set mode specified for the set, as shown in the following table.

Mode	Defaults
MODE IS CHAIN	Causes a default of NEXT DBKEY POSITION IS AUTO; the LINKED TO PRIOR clause causes a default of PRIOR DBKEY POSITION IS AUTO.
MODE IS INDEX	Causes a default of INDEX DBKEY POSITION IS AUTO. (Note that if the DBA codes NEXT or PRIOR, the schema compiler accepts the statement, but changes the specification to INDEX.)
MODE IS VSAM	Is not applicable to next and prior set pointers.

OMItted

Indicates no pointer will be maintained in the member record for the index. For a system-owned index, this means there are no index pointers in the member records. If you use this option for a system-owned index, you must also specify the MANDATORY AUTOMATIC set options.

index-dbkey-position

Assigns the sequential position of the index set pointer within the member record's prefix. *Index-dbkey-position* must be an integer in the range 1 through 8180. The default for the index pointer position depends on the set mode as shown in the table under the "Usage" topic

When assigning pointer positions manually, remember to specify this value if the set is an indexed set.

next-dbkey-position

Assigns the sequential position of the next set pointer within the member record's prefix. *Next-dbkey-position* must be an integer in the range 1 through 8180. The default for the next pointer position depends on the set mode as shown in the table under the "Usage" topic.

prior-dbkey-position

Assigns the sequential position of the prior set pointer within the member record's prefix. *Prior-dbkey-position* must be an integer in the range 1 through 8180. The default for the prior pointer position depends on the set mode as shown in the table under the "Usage" topic. Remember to specify this value if LINKED TO PRIOR is specified in the MODE clause.

LINked to OWNer

Links each member record of the named type in the set to the owner record.

OWNer dbkey POSition is *owner-dbkey-position*

Assigns the owner pointer position manually. *Owner-dbkey-position* represents a relative position in the member record's prefix to be used for storing the database key of the owner record of the set; it must be an unsigned integer in the range 1 through 8180. Do not specify this clause for:

- Indexed sets whose owner is SYSTEM
- Native VSAM sets

OWNer dbkey POSition is AUTo

Causes the schema compiler to automatically assign the owner pointer position within the member record's prefix when the schema is validated. AUTO is the default.

Until the schema description is validated, a DISPLAY or PUNCH of the set will indicate AUTO for the pointer position; after validation, these statements will indicate the actual sequential pointer position.

FOReign KEY is

For SQL access against a non-SQL defined database, identifies or removes a foreign key in the member record.

NULl

Removes a previously defined foreign key from the member record; if specified, the owner record must be defined without a primary key.

element-name

Identifies an element or a list of elements enclosed in parenthesis that identify the foreign key. The elements cannot be group elements and *must* match the data type and length of the corresponding element in the primary key.

NULlable

Indicates that the foreign key element can contain NULL values. To use this specification, the following rules apply:

- The membership option of the member record cannot be mandatory automatic
- The foreign key element cannot be a control key or subordinate to a control key in any sorted set
- The foreign key element cannot be a CALC key
- The foreign key element must be defined as NULLABLE in all primary/foreign key sets in which it is named

MANdatory

Specifies that occurrences of this record type cannot be disconnected from the set other than through an ERASE function. MANDATORY must be specified for native VSAM sets and index sets in which the index db-key position is omitted.

OPTional

Specifies that occurrences of this record type can be disconnected from the set without being erased.

Note: Either MANDATORY or OPTIONAL must be specified when including a member into a set.

AUTomatic

Specifies that occurrences of this record type are connected implicitly to the set as part of the STORE function. AUTOMATIC must be specified for native VSAM sets and index sets in which the index db-key position is omitted.

MANual

Specifies that occurrences of this record type are connected to the set only when the CONNECT function is issued.

Note: Either AUTOMATIC or MANUAL must be specified when including a member into a set.

key-expression

Identifies a sorted set. This clause is required if SORTED has been specified in the ORDER statement and is invalid for other set orders.

Note: In a multiple-member set, record occurrences are maintained in order within their record type, but are maintained in no predictable order with respect to records of other types within the set.

sort-element-name

Identifies the member record element(s) on whose values the set is to be sorted (that is, the sort control element).

Sort-element-name specifies the name of a group or elementary data item defined in an element description statement for the named member record type, with the following restrictions:

- No element named FILLER can be used in the sort control element.
- No element that redefines another element or is subordinate to an element that redefines another element can be used in the sort control element.
- No repeating element (that is, one defined with an OCCURS clause) and no element subordinate to a repeating element can be used in the sort control element.
- No element exceeding 256 bytes can be used in the sort control element.

Multiple *sort-element-name* values (each with its own order) can be coded, forming a compound sort control element and thereby allowing the member records to be sorted on more than 1 element within the record. The element names that make up the sort control element need not be contiguous within the member record. Note, however, that the combined lengths of the elements (as defined in the PICTURE and USAGE clauses of the ELEMENT substatement) must not exceed 256 bytes. Do not code multiple *sort-element-names* for native VSAM sets.

DBKey

For indexed sets only, specifies that the member record's database key is the set control element. Duplicates are not allowed.

ASCending

Sorts the specified sort-element or database key in ascending order. ASCENDING is the default. ASCENDING must be specified for native VSAM sets.

Note that if you specify ASCENDING before the KEY keyword, you cannot specify ASCENDING or DESCENDING anywhere else in *key-expression*.

DEScending

Sorts the specified sort-element or database key in descending order.

Note that if you specify DESCENDING before the KEY keyword, you cannot specify ASCENDING or DESCENDING anywhere else in *key-expression*.

NATural sequence

Indicates that the values of the key fields will be sorted and evaluated with negative values before positive values. By default, CA IDMS/DB sorts and evaluates the key fields using a standard collating sequence, which sorts information according to its hexadecimal representation.

Even if NATURAL SEQUENCE is specified, the schema compiler may use a standard sort sequence if an element in the sort key is a group element. If the data types of the elements subordinate to the group do not affect the natural sort sequence, CA IDMS/DB uses the natural sequence. Otherwise, it uses the standard sort sequence and issues a warning message.

Note: If STANDARD SEQUENCE is assumed and the CONTROL FIELDS will allow NATURAL SEQUENCE, NATURAL SEQUENCE will be selected. Control fields that are display will be set to NATURAL SEQUENCE.

UNCOMpressed

Applies to sorted indexed sets only and specifies that similar index entries will be maintained in their entirety.

COMpressed

Applies to sorted indexed sets only and specifies that similar index entries will be maintained in compressed form. COMPRESSED saves index space by compressing repeated characters and by causing like index entries to be stored in part: the initial like portion of the entry is stored once for all similar entries and only the different remaining portions are stored for each entry.

DUPlicates are

Specifies how CA IDMS/DB handles a record occurrence whose sort key duplicates an existing occurrence's sort key.

FIRst

Logically positions record occurrences before the occurrence(s) with the duplicated sort key. FIRST is not valid for native VSAM sets.

LASt

Logically positions record occurrences after the occurrence(s) with the duplicated sort key. LAST is not valid for native VSAM sets.

NOT allowed

Does not allow record occurrences with duplicate sort keys.

UNORDered

For native VSAM only, retrieves record occurrences in the order in which they were stored, regardless of the direction in which the set is being searched.

by DBkey

For MODE IS INDEX sets only, sorts record occurrences with duplicate key values by db-key.

DETails

Displays or punches the entire set description.

ALL

Displays or punches the entire set description.

NONe

Displays or punches only the set name.

Usage

Set Automatically Deleted if Owner Record is Deleted

If a set's *owner record* is deleted (by a DELETE RECORD statement), the set is automatically deleted. Additionally, the deleted record and set are deleted from all subschema descriptions associated with the current schema. But if a set's *member record* is deleted (by a DELETE RECORD statement), the set remains.

Explicitly deleting a set

To delete the set (if it has no other member records), use the DELETE SET statement. DELETE deletes the named set description from the data dictionary. Consequently, the set is removed not only from the current schema, but also from the descriptions of all subschemas associated with the current schema. No optional clauses are valid for DELETE operations.

Default automatic pointer assignments for owner records

A valid set description requires pointer positions for the owner record and for each member record.

The defaults for the owner pointer positions depend on the set's mode specification as shown in the following table. Positions for which "none" is indicated have no default and *must not* be specified; there is no such pointer position for these modes.

Set mode	NEXT	PRIOR
CHAIN (without LINKED TO PRIOR)	AUTO	none
CHAIN (with LINKED TO PRIOR)	AUTO	AUTO
VSAM	none	none
INDEX (with user-defined record type as owner)	AUTO	AUTO
INDEX (with SYSTEM as owner)	none	none

Default automatic pointer assignments for member records

A valid set description requires pointer positions for the owner record and for each member record. The defaults for the member record pointer positions depend on the set's mode specification as shown in the following table. Positions for which "none" is indicated have no default and *must not* be coded; these modes have no such pointer position.

Set mode	NEXT	PRIOR	INDEX
CHAIN (without LINKED TO PRIOR)	AUTO	none	none
CHAIN (with LINKED TO PRIOR)	AUTO	AUTO	none
VSAM	none	none	none
INDEX	none	none	AUTO

Unlinked indexes

An unlinked index is a system-owned index in which there are no index pointers in the member records. You specify an unlinked index by using the OMITTED option on the INDEX DBKEY POSITION clause of the MEMBER RECORD clause. Unlinked indexes provide the following advantages:

- You can load and rebuild unlinked indexes faster
- You can add or remove an unlinked index without restructuring the database, provided the control length of a compressed or variable length member record is not changed

However, unlinked indexes may increase processing overhead. For a more thorough discussion of the considerations, see the *CA IDMS Database Design Guide*.

The set options for an unlinked index must be MANDATORY AUTOMATIC.

Pointer positions in a record

Note that for a given record, each position must be assigned to only one set pointer, and the positions within the record must be contiguous.

SAME AS SET clause reduces coding

Because SAME AS SET copies an existing description, it can relieve the DBA of a considerable amount of coding. The DBA can create a base set description with SAME AS SET and code additional clauses to alter the description of the new set as desired.

Restrictions for SAME AS SET clause

SAME AS SET must not be specified for a set to which order, mode, owner, or member already is assigned. Consequently, placement of the SAME AS SET clause is restricted as follows:

- ADD operation—When used in an ADD operation, SAME AS SET must precede all other optional clauses.
- MODIFY operation—SAME AS SET cannot be used in a MODIFY operation unless the set was added with no optional clauses.

Don't change set pointers for existing databases

Do not change set pointers for existing databases. Use the NEXT DBKEY POSITION, PRIOR DBKEY POSITION, INDEX DBKEY POSITION only when adding new sets or when changing sets in a schema for which a database is not yet defined. If you must change set pointers, for example because a set is deleted, you must restructure your database.

Determine pointer positions before assigning pointers

For a given record, each position must be assigned to only one set pointer, and the positions within the record must be contiguous. When assigning positions manually, determine the pointer positions for all sets in the schema before coding set descriptions. This will avoid any conflicts (such as attempting to use the same position twice) and will speed up the mechanical process of adding set descriptions to the schema description.

Percentage offsets assist database maintenance

Of the page limiting options, OFFSET with percentage specifications is the most flexible. As a database grows and must eventually be expanded, the physical areas of the database must also be expanded. If the DBA originally expressed the owner record's page range as a percentage of an area, the range need not be respecified to fit the new physical area description; the runtime system will automatically assign the owner record to the same relative position in the new physical area.

Foreign keys and control length

The specification of a foreign key does not affect the control length of the member record. Foreign key elements may occur beyond the last control key even if the record is compressed or variable in length. However, if a foreign key element does begin after the control length and the record has a database procedure which will change the value of the foreign key field on a store or modify (for example, to convert it to upper case), then you should not use SQL INSERT statements to store new occurrences, nor SQL UPDATE statements to change the value of the foreign key. If you do use these statements, the value of the foreign key field *before* the procedure is executed will be used to validate the primary/foreign key relationship. This may cause the update to fail on a referential constraint violation or it may cause the member record to be associated with an incorrect owner.

Mixed page groups

Chained sets may not cross page group boundaries regardless of the MIXED PAGE GROUP BINDS ALLOWED option setting.

Examples

Minimum SET statement

The following example supplies the minimum SET statement required for the set to be a *valid* schema component:

```
add set name is insplan-rider
    order is last
    mode is chain
    owner is insplan
    member is rider
    mandatory automatic.
```

Defining a chained set

The following example specifies that new records in the COVERAGE-CLAIMS set are added immediately before the owner record, and that both next linkages (required) and prior linkages (optional) are used:

```
add set name is coverage-claims
    order is last
    mode is chain linked to prior
      .
      .
      .
```

Defining an indexed set

The following example identifies INDEX-JOB-TITLE as an indexed set; each of the set's bottom-level internal index records will contain 50 entries.

```
add set name is index-job-title
    order is sorted
    mode is index  block contains 50 keys
     .
     .
     .
```

Using SAME AS SET to reduce coding

As stated earlier, SAME AS SET copies all information from the copied set to the new set description; the schema compiler treats all subsequent clauses as MODIFY operations. In the following example, the MODE clause is treated as though the statement were a MODIFY SET statement; the statement creates the EMP-POSITION set, which is identical to EMP-POS set, except for its mode, and associates the new set with the current schema.

```
add set name is emp-position
    same as set emp-pos of schema testschm version is 1
    mode is chain linked to prior.
```

Calculating the page range of owner records

In the following example, physical area EMP-DEMO-REGION contains 1000 pages, numbered from 1 through 1000. At runtime, CA IDMS/DB will use the offset specified for the system owner record and store the record on pages 51 ((1000 * 5 * .01) + 1) through 1000.

```
... owner is system
         within area emp-demo-region
             offset 5 percent for 95 percent.
```

In the following example, ORG-DEMO-REGION contains 240 pages, numbered from 2001 through 2240. At runtime, CA IDMS/DB will store the owner record on pages 2041 (2001 + 40) through 2240.

```
... owner is system
        within area org-demo-region
            offset 40 pages for 200 pages.
```

Manually setting pointer positions

The following MEMBER clause example establishes the EMPOSITION record as a member of the JOB-POSITION set. EMPOSITION has NEXT and PRIOR pointers for this set in positions 1 and 2 of the record prefix; owner linkage is maintained, with the OWNER pointer in position 3 of the record prefix. Runtime operations for EMPOSITION are governed by the OPTIONAL disconnect and MANUAL connect option.

```
add set name is job-position
    order is next
    mode is chain  linked to prior
    owner is job
        next dbkey position is 1
        prior dbkey position is 2
    member is emposition
        next dbkey position is 1
        prior dbkey position is 2
        linked to owner
            owner dbkey position is 3
        optional manual.
```

Examples of sorted sets

The following example illustrates two sorted sets:

```
add set name is ooak-skill
  order is sorted
  mode is chain  linked to prior
  owner is ooak
    next dbkey position is 1
    prior dbkey position is 2
  member is skill
    next dbkey position is 1
    prior dbkey position is 2
    optional automatic
    key is skill-name ascending
      duplicates not allowed.
```

```
add set name is emp-expertise
  order is sorted
  mode is chain  linked to prior
  owner is employee
    next dbkey position is 10
    prior dbkey position is 11
  member is expertise
    next dbkey position is 4
    prior dbkey position is 5
    linked to owner
    owner dbkey position is 6
    mandatory automatic
    key is emp-expertise ascending
      duplicates first.
```

Examples of indexed sets

The following example defines sets similar to those in the previous example. in this example the sets are implemented as indexed sets:

```
add set name is ooak-skill
  order is sorted
  mode is index
    block contains 70 keys
  owner is system
  member is skill
    index dbkey position is 1
    optional automatic
    key is skill-name ascending
      compressed
      duplicates not allowed.

add set name is emp-expertise
  order is sorted
  mode is index
    block contains 50 keys
  owner is employee
    next dbkey position is 10
    prior dbkey position is 11
  member is expertise
    index dbkey position is 4
    linked to owner
      owner dbkey position is 5
    mandatory automatic
    key is emp-expertise ascending
      duplicates first.
```

Example of a multiple-member set

The following example illustrates a set with three member record types; the db-key position specification defaults to AUTO:

```
add set name is coverage-claims
    order is last  linked to prior
    mode is chain
    owner is coverage
    member is hospital-claim
        mandatory automatic
    member is non-hosp-claim
        mandatory automatic
    member is dental-claim
        mandatory automatic.
```

Primary/Foreign key usage:

Defining primary and foreign keys for network sets allows SQL to treat sets as referential constraints between network records. Incorporating a foreign key into the member record of a set and identifying the primary and foreign keys in the SET definition statement, allows standard application development tools that use JDBC and ODBC metadata functions, to discover the relationship between the network records in a set relationship. This also enables the use of standard SQL statements to INSERT, UPDATE, and DELETE rows in the owner and member records and eliminates the need for SQL syntax extensions and table procedures. If the department ID is defined in the EMPLOYEE record as DEPT-ID-4015, the following example shows how the DEPT-EMPLOYEE set can be defined as a referential set.

```
add set name is dept-employee
     .
     .
     .
    owner is department
     .
     .
     .
        primary key is calc
    member is employee
     .
     .
     .
        foreign key is dep-id-0415
```

Note: For more information about pointer positioning, system-owned index sets and system record types, and how CA IDMS/DB compresses index entries, see the *CA IDMS Database Design Guide*.

VALIDATE Statement

The VALIDATE statement verifies the relationships among all components of the schema that is current for update and sets the status of the schema to VALID (if no errors exist) or IN ERROR (if errors exist). CA IDMS/DB requires that a *valid* schema reside in the dictionary before any other activity involving the database can begin.

Only the schema compiler updates the status.

Syntax

```
►►─── VALIDATE ─────────────────────────────────────────────────────►◄
```

Usage

Effect of VALIDATE on schema

When the schema compiler validates the schema, it takes one of the following actions:

- If it finds no errors, the compiler sets the schema's status to VALID. VALID indicates that the schema is usable by other CA IDMS/DB software.
- If it finds errors, the compiler sets the schema's status to IN ERROR and issues messages indicating the exact nature of each error. The DBA uses these messages to determine what changes must be made for the schema to be valid. As long as the status is IN ERROR, other CA IDMS/DB software (such as the subschema compiler and utilities) cannot use the schema.

Must validate the schema following ADD and MODIFY

The schema compiler sets the schema's status to IN ERROR after the successful execution of an ADD SCHEMA or MODIFY SCHEMA statement. You must validate the schema to make it available to other CA IDMS/DB software.

VALIDATE resolves pointers

In addition to the verification described above, VALIDATE causes the schema compiler to resolve the pointer positions for which AUTO was specified in set description statements.

VALIDATE can be used at any time during schema definition

The VALIDATE statement can be used at any time to verify the relationships of schema components. For example, if the DBA has not yet defined sets, but wishes to verify the schema's record structures, VALIDATE can be used; in this case, however, the DBA should anticipate a warning for records whose location mode is VIA an undefined set.

REGENERATE Statement

The REGENERATE statement regenerates subschema load modules following changes to the schema that is current for update.

Syntax

```
►►─── REGenerate ─┬─ AFFected ─┬─ SUBSChemas ──────────────────────────►
                  └─ ALL ──────┘

 ►─┬─────────────────────────────────────────────┬──────────────────────►◄
   └─ as LOAd MODule Version version-number ─┘
```

Parameters

AFFected

Instructs the schema compiler to regenerate only those subschemas that have been affected by the schema modification.

ALL

Instructs the schema compiler to regenerate all subschemas associated with the current schema.

as LOAd MODule Version *version-number*

Specifies the version number to be assigned to the subschema load modules. *Version-number* must be an unsigned integer in the range 1 through 9999. The default is 1.

Note: Unlike other version numbers, the load module version number does not default to the current session option.

Usage

Effect of REGENERATE on Subschemas

In response to the REGENERATE statement, the schema compiler identifies each subschema that must be regenerated and verifies the relationships among the components of each identified subschema. Based on this verification, the schema compiler takes one of the following actions:

- If it finds no errors, the compiler invokes the subschema compiler to create subschema tables and store the tables as a load module in the dictionary load area (DDLDCLOD). The subschema is marked as VALID.
- If it finds errors, the compiler issues messages indicating the name of the invalid subschema and the exact nature of each error and marks the subschema IN ERROR. The DBA uses these messages to determine what changes must be made for the subschema to be valid and uses the subschema compiler to make any necessary changes to the subschema.

Using the subschema compiler to regenerate subschemas

Alternatively, after modifying and validating a schema, the DBA can use the subschema compiler (IDMSUBSC) to validate and regenerate subschemas. To regenerate subschemas, use either the schema compiler or the subschema compiler: using both causes needless duplication of processing. Note that if a subschema requires changes *in addition to* those necessitated by changes in the schema, the DBA need not use REGENERATE. The DBA can, after validating the schema, use the subschema compiler both to make the additional changes and to generate the new subschema load module.

Chapter 15: Subschema Statements

This section contains the following topics:

Overview

This chapter describes SUBSCHEMA statements. Syntax, parameter descriptions, usage information, and examples are presented for each statement.

Syntax Order

ADD/MODIFY syntax is presented first, followed by DELETE syntax. DISPLAY/PUNCH syntax is presented last.

Expansion Variables

Diagrams for expansion variables (indicated by underscore and italics) are shown at the end of the current syntax diagram. Expansions for common clauses are handled in a separate chapter, and those expansions are referenced in the parameter description.

Note: For more information about DISPLAY ALL syntax, see Chapter 11, “Compiler-Directive Statements”.

SUBSCHEMA Statement

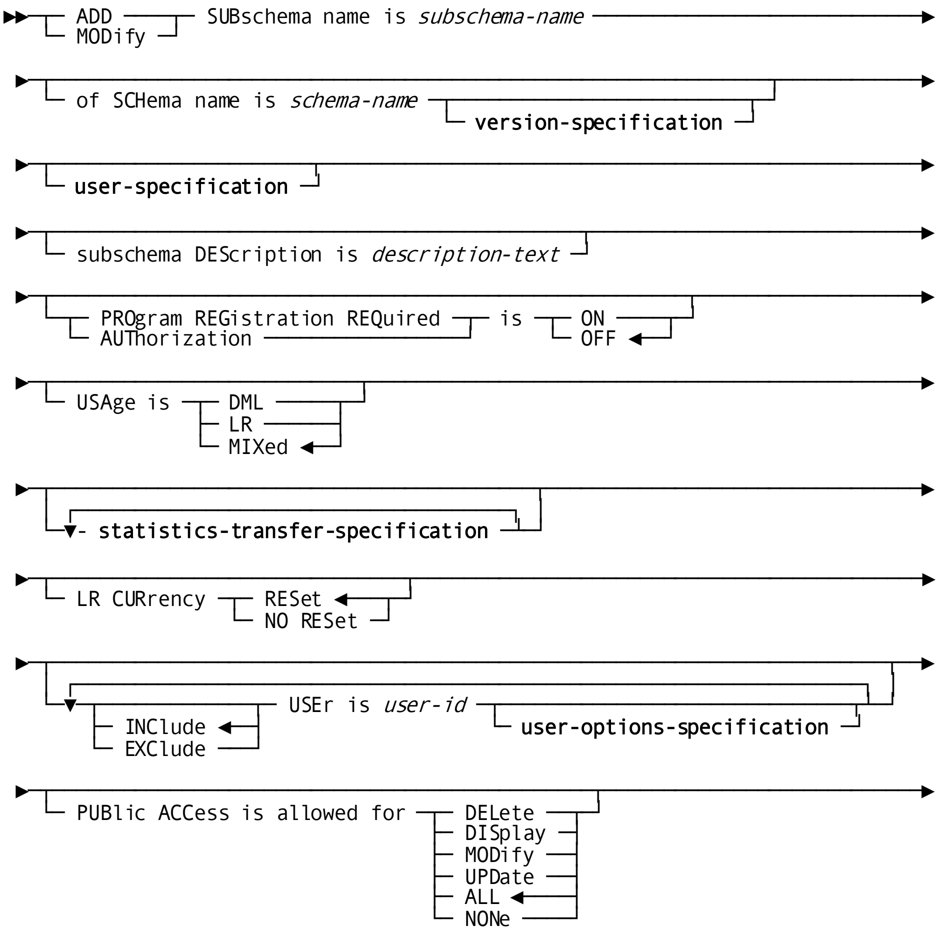
The SUBSCHEMA statements identify the subschema as a whole, and establish subschema currency as described in 9.7, "Establishing Schema and Subschema Currency".

In addition to the functions stated above, SUBSCHEMA statements can:

- Add, modify, delete, display, or punch a subschema description
- Establish security for the subschema
- Authorize users to issue specific verbs against the subschema

Syntax

Syntax: ADD/MODIFY SUBSCHEMA


ADD
MODify
SUBschema name is *subschema-name*
of SCHema name is *schema-name*
version-specification
user-specification
subschema DEScription is *description-text*
PROgram REGistration REQuired
AUThorization
is
ON
OFF
USAge is
DML
LR
MIXed
statistics-transfer-specification
LR CURrency
RESet
NO RESet
INClude
EXClude
USEr is *user-id*
user-options-specification
PUBlic ACCess is allowed for
DELete
DISplay
MODify
UPDate
ALL
NONe

```
 ►─┬─────────────────────────────────────────────────────────────────────┬─►
   │ ┌───────────────────────────────────────────────────────────────────┐ │
   └─▼─┬──────────────┬─ class-name is attribute-name ─┬───────────────────────┬┴─┘
       ├─ INClude ◄───┤                                 └─ TEXT is user-text ─┘
       └─ EXClude ────┘

 ►─┬──────────────────────────────────────────────┬────────────────────────────►◄
   └─┬─ COMments ────┬─┬─ comment-text ─┬─┘
     └─ comment-key ─┘ └─ NULl ─────────┘
```

Expansion of *statistics-transfer-specification*

```
 ►►─── TRAnsfer statistics to SUBschema name subschema-name ─────────────────►

 ►─┬───────────────────────────────────────────────────────────────┬──────────►
   └─ of SCHema name is schema-name ─┬───────────────────────────┬─┘
                                     └─ version-specification ─┘

 ►─┬───────────────────────────────────────────────────────────────────┬──────►◄
   └─ FOR PROgram name is program-name ─┬───────────────────────────┬─┘
                                        └─ version-specification ─┘
```

Syntax: DELETE SUBSCHEMA

```
 ►►─── DELete SUBschema name is subschema-name ──────────────────────────────►

 ►─┬───────────────────────────────────────────────────────────────┬──────────►
   └─ of SCHema name is schema-name ─┬───────────────────────────┬─┘
                                     └─ version-specification ─┘

 ►─┬─────────────────────────┬────────────────────────────────────────────────►◄
   └─ user-specification ─┘
```

Syntax: DISPLAY/PUNCH SUBSCHEMA

```
 ►►─┬─ DISplay ─┬─ SUBschema name is subschema-name ─────────────────────────►
    └─ PUNch ───┘

 ►─┬───────────────────────────────────────────────────────────────┬──────────►
   └─ of SCHema name is schema-name ─┬───────────────────────────┬─┘
                                     └─ version-specification ─┘

 ►─┬──────────────────────────────────────────────────────┬───────────────────►
   └─ PREpared by user-id ─┬─────────────────────────────┬─┘
                           └─ PASsword is password ─┘

 ►─┬────────────────────────────────────────────────────────────┬─────────────►
   │ ┌──────────────────────────────────────────────────────────┐ │
   │ │                 ┌─────────────────────────────────────┐ │ │
   └─▼─┬─ WITh ──────┬─▼─┬─ ALL COMment TYPes ──────────────┬─┴─┴─┘
       ├─ ALSo WITh ─┤   ├─ AREas ───────────────────────────┤
       └─ WITHOut ───┘   ├─ ATTributes ──────────────────────┤
                         ├─ COMments ────────────────────────┤
                         ├─ DEFinitions ─────────────────────┤
                         ├─ DETails ─────────────────────────┤
                         ├─ ELements ────────────────────────┤
                         ├─ HIStory ─────────────────────────┤
                         ├─ LRS ─────────────────────────────┤
                         ├─ PATh-groups ─────────────────────┤
                         ├─ PROgrams ────────────────────────┤
                         ├─ RECords ─────────────────────────┤
                         ├─ SETs ────────────────────────────┤
                         ├─┬─ USEr DEFINED COMments ─┬───────┤
                         │ └─ UDCs ──────────────────┘       │
                         ├─ USErs ───────────────────────────┤
                         ├─ ALL ─────────────────────────────┤
                         └─ NONe ────────────────────────────┘
```

```
►─┬────────────────────────┬───────────────────────────────────────────►
  └─ VERB ─┬─ ADD ─────────┘
           ├─ MODify ────┤
           ├─ DELete ────┤
           ├─ DISplay ───┤
           └─ PUNch ─────┘

►─┬──────────────────────┬─────────────────────────────────────────────►
  └─ AS ─┬─ COMments ─┬──┘
         └─ SYNtax ───┘

►─┬─────────────────────────────────────┬──────────────────────────────►◄
  └─ TO ─┬─ module-specification ─┬─────┘
         └─ SYSpch ───────────────┘
```

Parameters

SUBschema name is *subschema-name*

Identifies the subschema description to the dictionary. *Subschema-name* specifies the name of the subschema. *Subschema-name* must be a 1- to 8-character alphanumeric value.

of SCHema name is *schema-name*

Associates the subschema with a previously compiled schema. *Schema-name* is the name of a valid schema for which the named subschema represents a program view. This clause is required for ADD operations; it is required for all other operations if the subschema name is not unique in the dictionary.

version-specification

Specifies the version number of the schema. The version number defaults to the current session option for existing versions.

Note: Expanded syntax for *version-specification* is presented in Chapter 13, “Parameter Expansions".

user-specification

Identifies the user using the subschema description.

Note: Expanded syntax for *user-specification* is presented in Chapter 13, “Parameter Expansions".

subschema DEScription is *description-text*

Optionally specifies a name that is more descriptive than the 8-character subschema name required by CA IDMS/DB, but can be used to store *any* type of information; SUBSCHEMA DESCRIPTION is purely documentational. *Description-text* is a 1- to 40-character alphanumeric field; if it contains spaces or delimiters, it must be enclosed in quotes.

For CA OLQ users, the descriptive information appears on the CA OLQ screen to select subschemas.

PROgram REGistration REQuired/AUThorization is ON

Specifies that programs must be registered with the named subschema in order to be compiled against the subschema. AUTHORIZATION is a synonym for PROGRAM REGISTRATION REQUIRED.

To register a program with a subschema, use the IDD DDDL PROGRAM statement. A program naming the subschema is not eligible for compilation by the DML precompilers unless it is registered with the subschema.

PROgram REGistration REQuired/AUThorization is OFF

Specifies that programs do not have to be registered with the named subschema to be compiled against the subschema. AUTHORIZATION is a synonym for PROGRAM REGISTRATION REQUIRED. OFF is the default. Any program naming the subschema can be compiled by the DML precompilers.

USAge is DML

Specifies that programs using the subschema can access database records only. Attempts to access logical records will result in the return of an error-status code of 2010 to the requesting program.

USAge is LR

Specifies that programs using the subschema can access logical records only. Attempts to access database records will result in the return of an error-status code of *nn*10 to the requesting program.

USAge is MIXed

Specifies that programs using the subschema can access both database records and logical records. MIXED is the default.

statistics-transfer-specification

Transfers compile-time program statistics from the current subschema to another subschema. Statistics can be transferred for all programs associated with the current subschema or for a specific program. Statistics can be viewed in standard dictionary (DREPORTs) activity reports (see the *CA IDMS Reports Guide*), as follows:

- Area statistics by area and by program
- Set statistics by set and by program
- Record statistics by record and by program
- Logical record statistics by logical record and by program

TRAnsfer statistics to SUBschema name *subschema-name*

Identifies the subschema to receive the transferred statistics.

of SCHema name is *schema-name*

Identifies the schema with which the subschema receiving the transferred statistics is associated; this clause is required if *subschema-name* is not unique.

version-specification

Uniquely qualifies *schema-name* with a version number. The default is the current session option for existing versions.

Note: Expanded syntax for *version-specification* is presented in Chapter 13, “Parameter Expansions".

FOR PROgram name is *program-name*

Identifies a program for which statistics have been collected under the current subschema. The statistics for and registration of the named program are transferred to the subschema named in the TRANSFER STATISTICS clause. If this clause is omitted, the statistics for all programs associated with the subschema will be transferred.

version-specification

Uniquely qualifies *program-name* with a version number. The default is the current session option for existing versions.

Note: Expanded syntax for *version-specification* is presented in Chapter 13, “Parameter Expansions".

LR CURrency RESet

For subschemas containing logical-record definitions, specifies that the CA IDMS Logical Record Facility (LRF) is to reset currency and restore the logical record's program variable storage area before iterating a path. RESET is the default.

LRF sets the currency to that which existed at the termination of the previous execution of the path and restores the logical record's variable storage area with the records obtained during the previous execution of the path. LRF resets currency by issuing FINDs by DBKEY for all logical-record elements previously located up to, but not including, that element at which iteration is to commence. LRF restores storage by additionally issuing GETs for those elements retrieved as well as located during the previous execution of the path.

LR CURrency NO RESet

Specifies that LRF is not to reset currency or restore variable storage.

INClude USEr is *user-id*

Associates a user with the subschema description. *User-id* must be the name of a user as defined in the dictionary.

user-options-specification

Registers the user to access the subschema description, places security on the subschema description, and documents the user's association with the subschema. The options available with this clause are valid for INCLUDE only.

Note: Expanded syntax for *user-options-specification* is presented in Chapter 13, “Parameter Expansions".

PUBlic ACCess is allowed for

Specifies which operations are available, for the current subschema and its components, for public access (that is, to all users who can sign on to the subschema compiler. When coded, the keyword ALLOWED can be abbreviated to no fewer than four characters (ALLO).

DELete

Allows unregistered users to DELETE, DISPLAY, and PUNCH the subschema and its components.

DISplay

Allows unregistered users to DISPLAY and PUNCH the subschema and its components.

MODify

Allows unregistered users to MODIFY, DISPLAY, and PUNCH the subschema and its components.

UPDate

Allows unregistered users to ADD, MODIFY, DELETE, DISPLAY, and PUNCH the subschema and its components. Unlike ALL, UPDATE does not allow unregistered users to change the subschema's PUBLIC ACCESS specification.

ALL

Allows unregistered users to ADD, MODIFY, DELETE, DISPLAY, and PUNCH the subschema and its components. Additionally, ALL allows all users to change the subschema's PUBLIC ACCESS specification, thus enabling them to change security for the subschema.

NONe

Prohibits unregistered users from accessing the subschema.

INClude *class-name* is *attribute-name*

Provides a way for the DBA to classify the subschema for documentational purposes by associating an attribute with the subschema.

Class-name must be the name of a class as defined in the dictionary through the IDD DDDL compiler. If the dictionary entry for the class specifies that attributes must be added manually, *attribute-name* must be the name of an attribute already associated with *class-name*; if not, *attribute-name* can be any 1- to 40-character value, enclosed in quotes if it contains spaces or delimiters.

Note: For instruction in defining classes and attributes, see the *CA IDMS IDD DDDL Reference Guide*.

EXClude *class-name* is *attribute-name*

Dissociates an attribute with the subschema. *Class-name* must be the name of a class for which an attribute already is associated with the subschema; *attribute-name* names the attribute in order to be dissociated from the subschema.

TEXT is *user-text*

On INCLUDE *class-name* operations, supplies additional documentation of the assignment of a specific attribute to the subschema. *User-text* is 1 to 40 characters of text; if it contains spaces or delimiters, it must be enclosed in quotes.

COMments/*comment-key* is *comment-text*/NULl

Provides a way for the DBA to maintain comments about the subschema. *Comment-key* is the value assigned in the USER DEFINED COMMENTS clause of the IDD DDDL MODIFY ENTITY statement. NULl disassociates text from the current subschema.

Note: Coding rules for *comment-text* are presented in 10.5.4, "Coding Comment Text".

ALL COMment TYPes

Displays and punches all comment entries (COMMENTS, DEFINITIONS, ELEMENT DEFINITIONS, CULPRIT HEADERS, OLQ HEADERS, REMARKS, and user-defined comment keys) associated with the requested subschema.

AREas

Displays and punches all areas included in the subschema.

ATTributes

Displays and punches all classes and attributes assigned to the subschema.

COMments

Displays and punches all COMMENTS clauses included both in the SUBSCHEMA statement and in all logical-record definitions in the subschema.

DEFinitions

Displays and punches all definitions associated with the subschema.

DETails

Displays and punches the following information about the subschema:

- The AUTHORIZATION clause specified for the subschema
- The USAGE clause specified for the subschema
- The LR CURRENCY clause specified for the subschema

ELements

Displays and punches the following information:

- When LRS and DETAILS are also specified, database records contained in a logical record definition
- When RECORDS and DETAILS are also specified, elements (fields) previously specified in a subschema record definition

HIStory

Displays and punches the date and time that the subschema was created or last modified.

LRS

Displays and punches all logical records included in the subschema.

PATh-groups

Displays and punches all logical-record path groups included in the subschema.

PROgrams

Displays and punches all programs associated with the subschema.

RECords

Displays and punches all database records included in the subschema.

SETs

Displays and punches all sets included in the subschema.

USEr DEFINED COMments/UCDS

For subschema with user-defined comment keys only, displays and punches all user-defined comment keys associated with the requested subschema.

USErs

Displays and punches all users associated with the subschema, including the REGISTRATION, RESPONSIBILITY, and PUBLIC ACCESS clauses.

ALL

Displays and punches the entire subschema description.

NONe

Displays and punches only the subschema name and associated schema name and version number.

Usage

Effect of ADD on Subschema

ADD creates a new subschema source description in the dictionary. Default values established through the SET OPTIONS statement can be used to supplement the user-supplied description.

ADD also sets the subschema's status to IN ERROR. The status must be set to VALID before a subschema load module can be generated; a load module must be generated before programs can use the subschema to access the database.

Effect of MODIFY on Subschema

MODIFY modifies an existing subschema source description in the dictionary. All clauses associated with an ADD operation can be specified for MODIFY operations.

MODIFY also sets the subschema's status to IN ERROR. The status must be set to VALID before a subschema load module can be generated. Note, that if modification involves the following changes, and if the subschema already has a load module, a new load module need not be produced:

- Documentation
- Program registration
- Statistics transfer
- Users included or excluded
- Public access

Effect of DELETE on Subschema

DELETE deletes an existing subschema source description from the dictionary. The subschema load module (if any) remains intact, unless the SET OPTION statement specifies DELETE IS ON, in which case the subschema compiler:

- Logically deletes version 1 of the subschema load module from the load area of the dictionary (load modules qualified by another version number must be explicitly deleted).
- Automatically erases version 1 of any PROG-051 dictionary record occurrence associated with the subschema load module, provided the record was built by the subschema compiler and is not related to any other entity type in the dictionary.

SUBSCHEMA Statement Defines Its Use by Program

The SUBSCHEMA statement defines the following information about its use by programs:

- Program authorization—The SUBSCHEMA statement specifies whether programs using the subschema must be registered with the subschema (by means of the IDD DDDL compiler) in the dictionary in order to be eligible for compilation by the CA IDMS Data Manipulation Language (DML) precompilers.
- DML usage—The SUBSCHEMA statement specifies whether programs using the subschema can issue only DML requests, only logical-record DML requests, or both.

The SUBSCHEMA statement can also be used to transfer the statistics (such as database access statistics) for the named subschema to another subschema.

ADD Interpreted as MODIFY

If, on an ADD operation, a subschema of the same name within the same schema already exists in the dictionary, the action taken by the subschema compiler varies depending on the current session option for DEFAULT:

- If DEFAULT IS ON is specified, the subschema compiler interprets the ADD as a MODIFY for the named subschema.
- If DEFAULT IS OFF is specified, the subschema compiler issues an error message and terminates processing of the ADD SUBSCHEMA statement. Note that, in this case, subschema currency will be null for subsequent statements.

User-Specification Required for Secured Subschemas

If the *user-specification* clause is not used, *user-id* and *password* default to the current session options.

User-specification is used when the subschema compiler checks security. If either the subschema compiler or the specific subschema is secured, the compiler rejects the operation unless it finds the name and password of an authorized user in one of the following places:

- The SUBSCHEMA statement
- The current session value

Note: For a detailed description of security, see the *CA IDMS Security Administration Guide*.

Transferring Statistics for Some, But Not All, Programs

To transfer statistics for multiple (but not all) programs, repeat the TRANSFER STATISTICS clause for each program. To transfer statistics for all programs registered with the subschema, include a single TRANSFER STATISTICS clause that does not specify a program name.

Existing User Registration Replaced by New One

When modifying a user's registration, the option specified in the REGISTERED FOR clause replaces the previous specification. In the following example, the second REGISTERED FOR clause removes BARBER's ability to delete subschema empss01.

```
add subschema empss01
    user is barber
    registered for update.

mod subschema empss01
    user is barber
    registered for modify.
```

Existing User Responsibility Replaced by New One

When modifying a user's responsibility documentation, the option specified in the RESPONSIBLE FOR clause replaces the previous specification. In the following example, the second RESPONSIBLE FOR clause removes CREATE from BAKER's documentation of responsibilities.

```
add subschema empss02
    user is baker
    responsible for creation and update.

mod subschema empss02
    user is baker
    responsible for update.
```

Registered Users Can Perform Non-Public Access Operations

To perform any operation not available for public access, the user must be registered for that operation in the current subschema. Registered users can also perform operations available for public access.

At Least One User Must Be Registered for ALL

When a subschema is added to the dictionary, public access defaults to ALL and cannot be changed until at least one user is registered for ALL operations. The first registration of a user for ALL operations changes public access to NONE. Note that the last user with ALL registration cannot be excluded from the subschema description until public access is changed to ALL. Thus the subschema compiler ensures that no inaccessible subschema description exists in the dictionary. The following example illustrates the various stages of public access:

```
add subschema empss01.
```

Public access defaults to all.

```
mod subschema empss01.
    user is mjj
      registered for all
    public access is modify.
```

Public access changes from NONE to MODIFY with PUBLIC ACCESS is MODIFY.

```
mod subschema empss01.
    exclude user mjj.
```

This statement is not possible. Public access must first be changed to all.

Assigning Text to a Comment Key

Before entering *comment-text* for a *comment-key* in the COMMENTS clause, the comment key must have been previously defined in the USER DEFINED COMMENTS clause of the IDD DDDL MODIFY ENTITY statement.

Before specifying EXCLUDE in the USER DEFINED COMMENTS clause of an IDD DDDL MODIFY ENTITY statement, you must first specify NULL for the comment key in the SUBSCHEMA COMMENTS clause.

Examples

Minimum SUBSCHEMA Statement

The following example supplies the minimum SUBSCHEMA statement required for the purpose of later establishing a functional subschema:

```
add subschema name is dehss01
  of schema empschm version 100.
```

Securing the subschema for LRF Usage

This example modifies subschema DEHSS01 so that any program that uses the subschema must first be registered. It also designates that these programs can access logical records only.

```
mod subschema dehss01
   program registration is on
   usage is lr .
```

Registering a User For All Operations

This example indicates user DEH has authority to perform all basic entity operations and to issue the PUBLIC ACCESS clause. All other users are allowed to display or punch the subschema.

```
mod subschema dehss01
   include user deh
       registered for all
   public access is allowed for display.
```

Documenting Subschema Revisions

In the following example, the DBA documents subschema revisions and the purpose of those revisions; note that the DBA first defined revision number as a class in the dictionary:

```
modify subschema name is culss01
    prepared by dba  password is tennis
    revision number is '6.5'
        text is 'accommodate new billing restrictions'.
```

Note: For more information about when to specify LR CURRENCY RESET or NO RESET, see the *CA IDMS Logical Record Facility Guide*.

AREA Statement

The AREA statements identify a subschema area. Depending on the verb and options coded, the AREA statements can also:

- Copy an area description from the schema with which the current subschema is associated
- Determine the usage modes in which programs using the current subschema can ready the area
- Determine the default usage mode for programs that do not issue READY statements
- Delete an area from the subschema
- Display or punch a subschema area

The subschema compiler applies AREA statements to the current subschema.

Note: For more information about subschema currency, see the *CA IDMS Security Administration Guide*.

Syntax

Syntax: ADD/MODIFY AREA

```
►►─┬─ ADD ────┬─ AREa name is area-name ──────────────────────────────────────►
   └─ MODify ─┘

 ►─┬─────────────────────────────────────────────────────────────────────┬───►
   │ ┌─────────────────────────────────────────────────────────────────┐ │
   └─▼─┬──────────────┬─┬─ UPDate ────┬─ is ─┬─ ALLowed ◄──────────────┴─┘
       ├─ EXClusive ─┤ └─ RETrieval ─┘      └─ NOT ALLowed ─┘
       ├─ PROtected ─┤
       └─ SHAred ────┘

 ►─┬──────────────────────────────────────────────────────────────────────┬──►◄
   └─ DEFault USAge is ─┬─┬──────────────┬─┬─ UPDate ────┬─┬──────────┬─┬─┘
                        │ ├─ EXClusive ─┤ └─ RETrieval ─┘ └─ FORce ──┘ │
                        │ ├─ PROtected ─┤                               │
                        │ └─ SHAred ────┘                               │
                        └─ NULl ◄───────────────────────────────────────┘
```

Syntax: DELETE AREA

```
►►─── DELete AREa name is area-name ──────────────────────────────────────►◄
```

Syntax: DISPLAY/PUNCH AREA

```
►►─┬─ DISplay ─┬─ AREa name is area-name ─────────────────────────────────────►
   └─ PUNch ───┘

 ►─┬───────────────────────────────────────────┬──────────────────────────────►
   │ ┌───────────────────────────────────────┐ │
   └─▼─┬─ WITh ───────┬─▼┬─ DETails ─┬───────┴─┘
       ├─ ALSo WITh ─┤  ├─ ALL ─────┤
       └─ WITHOut ───┘  └─ NONe ────┘

 ►─┬───────────────────────────┬──────────────────────────────────────────────►
   └─ VERB ─┬─ ADD ───────┬────┘
            ├─ MODify ────┤
            ├─ DELete ────┤
            ├─ DISplay ───┤
            └─ PUNch ─────┘

 ►─┬────────────────────────────┬─────────────────────────────────────────────►
   └─ AS ─┬─ COMments ─┬────────┘
          └─ SYNtax ───┘

 ►─┬─────────────────────────────────────────┬────────────────────────────────►◄
   └─ TO ─┬─ module-specification ─┬─────────┘
          └─ SYSpch ───────────────┘
```

Parameters

AREa name is *area-name*

Identifies an area description. *Area-name* must be the name of an area defined in the schema with which the current subschema is associated.

UPDate

Specifies an area ready mode of UPDATE. Run units can ready the area for shared update, protected update, or exclusive update.

RETrieval

Specifies an area ready mode of RETRIEVAL. Run units can ready the area for shared retrieval, protected retrieval, or exclusive retrieval.

EXClusive

Specifies an area ready mode of EXCLUSIVE UPDATE or EXCLUSIVE RETRIEVAL.

PROtected

Specifies an area ready mode of PROTECTED UPDATE or PROTECTED RETRIEVAL.

SHAred

Specifies an area ready mode of SHARED UPDATE or SHARED RETRIEVAL.

is ALLowed

Specifies that run units using the current subschema can ready the area in the specified ready mode. ALLOWED is the default.

is NOT ALLowed

Specifies that run units using the current subschema *cannot* ready the area in the specified ready mode.

DEFault USAge is

Specifies the default ready mode, if any, in which the named area is to be readied for programs using the current subschema.

UPDate

Specifies the default ready mode is UPDATE. The area can be readied in SHARED UPDATE, EXCLUSIVE UPDATE, or PROTECTED update.

RETrieval

Specifies the default ready mode is RETRIEVAL. The area can be readied in SHARED RETRIEVAL, EXCLUSIVE RETRIEVAL, or PROTECTED RETRIEVAL.

EXClusive

Specifies the default ready mode is either EXCLUSIVE UPDATE or EXCLUSIVE RETRIEVAL.

PROtected

Specifies the default ready mode is either PROTECTED UPDATE or PROTECTED RETRIEVAL.

SHAred

Specifies the default ready mode is either SHARED UPDATE or SHARED RETRIEVAL.

FORce

Specifies that the area is automatically readied even if explicit READY statements for other areas have already been issued. If this parameter is omitted, then automatic ready is disabled after any explicit READY statement.

Note: For more information about FORCE, see the Usage (see page 466) section.

NULl

Specifies that programs accessing this area must issue an explicit READY statement for the area. NULL is the default.

DETails

Displays and punches the ready modes in which the area can or cannot be readied and the default ready mode in which the area will be readied for programs using the current subschema.

ALL

Displays and punches the entire area description.

NONe

Displays and punches only the name of the area.

Usage

Effect of ADD on Areas

ADD copies the area description from the schema description into the subschema description.

Effect of DELETE on Areas

DELETE removes the area from the current subschema description in the dictionary; the area remains associated with the schema.

AREA Statement Determines How Programs Can Ready the Area

ADD and MODIFY AREA operations can restrict the ready modes in which programs using the current subschema can ready the area, and can specify a default ready mode in which the area will be readied for programs using the current subschema.

The UPDATE (RETRIEVAL) IS ALLOWED clause can be repeated for as many different ready modes as required.

Specify Default Ready Mode for All Subschema Areas or use the FORCE Option

If a program issues an explicit READY for one area, it must issue an explicit READY for all areas to be accessed unless the areas use the default usage mode with the FORCE option. The automatic READY mechanism is turned off as soon as one area is readied explicitly by a program and then only areas using the default usage option FORCE are automatically readied.

Considerations for Using the FORCE Option with ADS Dialogs

The FORCE option is provided to allow application changes to be deferred when a record in an index set is moved to a different area. This type of change might be done to implement a Mixed Page Group Index Set. When the FORCE option is used on an AREA referenced by ADS dialogs having extended run units, those dialogs must be modified in some manner, as indicated in the scenarios and workarounds outlined below.

Three types of issues can occur with respect to the use of the FORCE option with areas that ADS dialogs access using an extended run unit:

1. **Negative impact on performance**

   Most ADS dialogs READY all areas for RETRIEVAL, therefore the default usage modes are often defined as RETRIEVAL. However, UPDATE dialogs require one or more areas to be readied for UPDATE. In cases where a new area is defined using the FORCE option and the subschema is referenced in one or more UPDATE dialogs, the RETRIEVAL/UPDATE clause for the new area must be set to UPDATE to allow updating the index set. Applying this setting changes all RETRIEVAL dialogs which reference that subschema area to UPDATE, which can cause degradation in performance, or in some cases may cause deadlocks to occur due to increased locking.

   Workaround

   Identify the dialogs that UPDATE the index (there should not be too many). Change the UPDATE dialogs by one of the two following methods:

   - Add a READY UPDATE statement in the process code (instead of using the Area FORCE option).
   - Use a new tailored subschema that readies the new area using the FORCE and UPDATE options.

   This workaround only functions if the run units are not extended.

2. **Abends in dialogs using extended run units.**

   This issue can occur in situations such as this example:

   An update dialog readies one area for UPDATE and another for RETRIEVAL. This dialog then LINKs to a dialog that does not ready the second area, but also readies the first area for UPDATE to allow updates to an indexed set that the upper-level dialog does not use. For efficiency, ADS keeps the run unit open through both dialogs. If one record of the indexed set is moved from the UPDATE area to the RETRIEVAL area, the lower-level dialog needs UPDATE access to the new area.

   As the run unit is rebound only when the dialogs are recompiled, the second run unit does not have UPDATE access to the second area. ERROR-STATUS abends such as 0801 and 1209 can occur.

   Workaround

   Change the subschema to ready its second area for UPDATE using the FORCE option. Be aware that doing so can cause the locking problems described in the preceding issue.

   Note: Using a tailored subschema for the lower-level dialog is not an option because the run unit would no longer be extended.

3. **Incomplete recovery after an abend of an extended run unit**

   When two dialogs use the same subschema with extendable attributes, the run unit is extended if one dialog LINKs to the other. When an indexed set is modified to span page groups, then neither of these dialogs is able to access the new area unless the dialogs are recompiled or the subschema is modified to FORCE a READY for the new area.

   Problems can occur in the situation when a subschema is changed to add a new area using the FORCE and UPDATE options. When a lower-level dialog in an extended run unit is recompiled, its RAT (READY AREA TABLE) is changed to access the new area and the run unit from the higher-level dialog will no longer be extended. In normal situations, the dialog performs as before, except that multiple run units are bound and finished during each execution of the transaction. If an abend occurs, only part of the transaction is rolled back because multiple run units were bound.

   Workaround

   Recompile all dialogs in the run unit thread if the FORCE option is used on an AREA referenced by ADS dialogs having extended run units.

Example

This example adds area EMP-DEMO-REGION to the current subschema. EMP-DEMO-REGION can be readied for SHARED UPDATE or SHARED RETRIEVAL. The default ready mode is SHARED RETRIEVAL.

```
add area name is emp-demo-region
   shared update is allowed
   default usage is shared retrieval.
```

Note: For more information about area ready modes and ready options, see the *CA IDMS Navigational DML Programming Guide*.

RECORD Statement

The RECORD statements identify a subschema record. Depending on the verb and options coded, the RECORD statements can also:

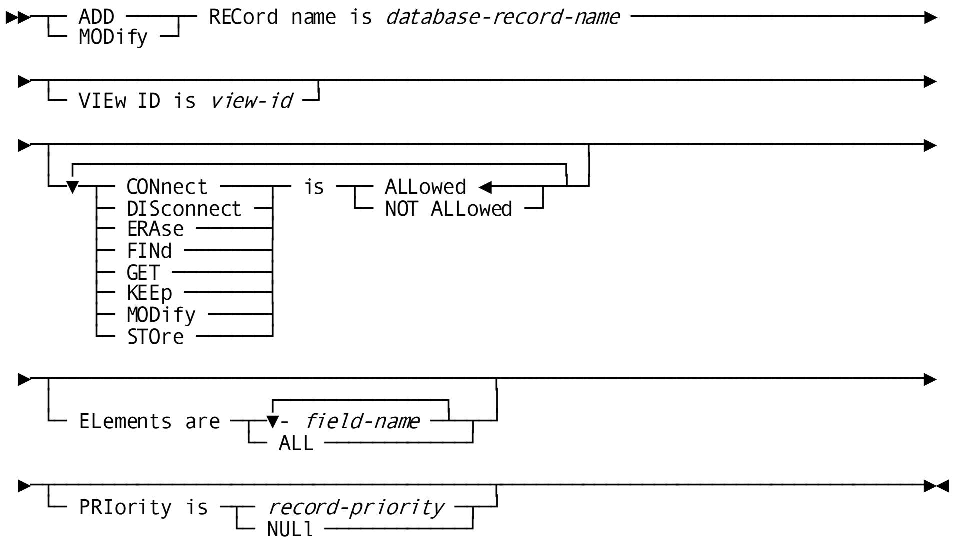
- Copy a record description from the schema with which the current subschema is associated
- Define a subschema view of the record; a subschema view determines:
  - Which record elements can be accessed through the subschema
  - Which DML verbs can be issued against the record
- Establish a priority, within the subschema, for the record
- Delete a record from the subschema
- Display or punch a subschema record description

The subschema compiler applies RECORD statements to the current subschema.

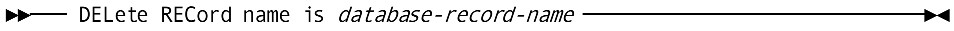
Note: For an explanation of subschema currency, see 9.7, “Establishing Schema and Subschema Currency”.

Syntax

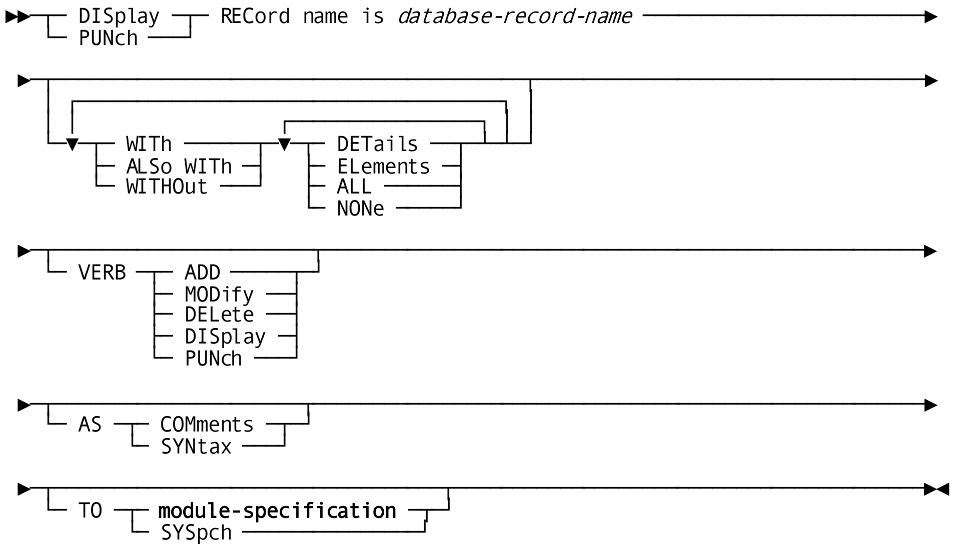
Syntax: ADD/MODIFY RECORD



Syntax: DELETE RECORD



Syntax: DISPLAY/PUNCH RECORD



Parameters

RECord name is *database-record-name*

Names a database record described in the schema with which the current subschema is associated. *Database-record-name* can be a record synonym of a schema record, in which case it must not exceed 16 characters.

VIEw ID is *view-id*

Copies a predefined view of the record description into the subschema, or it defines a view being created by the current subschema for this record description.

If *view-id* exists in the dictionary, that view is copied into the subschema. In this case, *view-id* must be the identifier of a database record placed in the dictionary by previous execution of the subschema compiler for another subschema, or by the IDD DDDL compiler (via DDDL RECORD entity-type syntax). If *view-id* does not already exist in the dictionary, it defines a new view of *database-record-name* in the dictionary, and it can subsequently be used for another subschema compiled under any schema that copies the same database record.

View-id must be a 1- to 32-character alphanumeric value. Additionally, it must be unique for the record, but need not be unique among all records defined in the dictionary.

CONnect

Specifies that programs using the current subschema can or cannot issue CONNECT commands against *database-record-name*.

DISconnect

Specifies that programs using the current subschema can or cannot issue DISCONNECT commands against *database-record-name*.

ERAse

Specifies that programs using the current subschema can or cannot issue ERASE commands against *database-record-name*.

FINd

Specifies that programs using the current subschema can or cannot issue FIND commands against *database-record-name*.

GET

Specifies that programs using the current subschema can or cannot issue GET commands against *database-record-name*.

KEEp

Specifies that programs using the current subschema can or cannot issue KEEP commands against *database-record-name*.

MODify

Specifies that programs using the current subschema can or cannot issue MODIFY commands against *database-record-name*.

STOre

Specifies that programs using the current subschema can or cannot issue STORE commands against *database-record-name*.

is ALLowed

Specifies that the program using the current subschema can issue the specified DML function against the database record. ALLOWED is the default.

is NOT ALLowed

Specifies that the program using the current subschema cannot issue the specified DML function against the database record.

ELements are *field-name*

Identifies the schema-defined fields to be included in the subschema description of *database-record-name*. *Field-name* must identify a field defined for *database-record-name* in the schema associated with the current subschema. (This is also true if *database-record-name* is a synonym.)

Note: For more information about using this clause, see "Usage" in this section.

ELements are ALL

Includes all schema-defined fields to be in the subschema description of *database-record-name*.

PRIority is *record-priority*

Specifies a priority to be assigned to the record in the runtime subschema tables. The PRIORITY clause is used to sequence record descriptions according to their priority in the subschema tables. For example, heavily-used records should receive a higher priority than less-frequently used records.

Record-priority is an unsigned integer in the range 0 through 9999, where 0 represents the lowest priority and 9999 represents the highest priority. If the PRIORITY clause is not included for a record, the record's sequence in the runtime subschema tables will correspond to that in which it was included in the current subschema. Records with the same priority are organized in the order included, within priority.

PRIority is NULl

Specifies that this record description is to be assigned no priority (that is, it will be placed at the end of the subschema tables).

DETails

Displays and punches the elements, access restrictions, view, and priority defined in the subschema record description. Note that only those elements previously specified in a subschema RECORD statement are displayed.

ELements

When DETAILS is also specified, displays and punches the elements specified in the ELEMENTS ARE clause of the subschema record definition.

ALL

Displays and punches the entire record description.

NONe

Displays and punches only the name of the record.

Usage

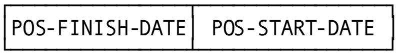
Effect of ADD on Records

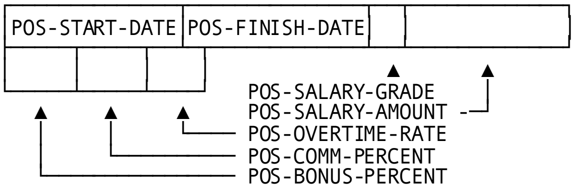
ADD copies the record description from the schema description into the subschema. The record can be copied into the subschema with its primary name or with any of its synonyms.

Note: A record description can be copied only once into a subschema, regardless of the number of record synonyms that exist for that record.

The following illustrates the use of the ADD RECORD statement. The left-hand side illustrates the original schema record description. The right-hand side illustrates a subschema record description, a subset of the schema.

```
SCHEMA                                      SUBSCHEMA

ADD RECORD NAME IS EMPOSITION            ADD RECORD NAME IS EMPOSITION
   LOCATION MODE IS VIA EMP-POSITION SET      STORE IS NOT ALLOWED
   WITHIN EMP-DEMO-REGION AREA.               ERASE IS NOT ALLOWED
   02 POS-START-DATE.                         ELEMENTS ARE
      03 POS-START-YEAR   PIC 99.                 POS-FINISH-DATE
      03 POS-START-MONTH  PIC 99.                 POS-START-DATE.
      03 POS-START-DAY    PIC 99.
   02 POS-FINISH-DATE.
      03 POS-FINISH-YEAR  PIC 99.
      03 POS-FINISH-MONTH PIC 99.
      03 POS-FINISH-DAY   PIC 99.
   02 POS-SALARY-GRADE    PIC 99.
   02 POS-SALARY-AMOUNT   PIC S9(7)V99 COMP-3.
   02 POS-BONUS-PERCENT   PIC S999 COMP-3.
   02 POS-COMM-PERCENT    PIC S999 COMP-3.
   02 POS-OVERTIME-RATE   PIC S999 COMP-3.
```





Effect of DELETE on Records

DELETE removes the record from the current subschema description in the dictionary; the record remains associated with the schema.

How DELETE RECORD Affects Set Definitions

If the record owns a subschema set, DELETE RECORD deletes the set. If the record is a member of a subschema set, DELETE RECORD has no effect on the set.

The subschema DELETE RECORD statement does not affect the schema description of sets.

How ELEMENTS and VIEW ID Clauses Determine the Record Description

The combination of the ELEMENTS clause specification and the VIEW ID clause specification determines which fields are copied into the subschema description of *database-record-name*. The following table lists the possible combinations of the ELEMENTS clause and VIEW ID clause specifications and the resulting subschema view of the record.

	No VIEW ID Clause	VIEW ID Clause
No ELEMENTS Clause	All schema-defined fields	Fields defined for record identified by view ID
ELEMENTS ARE ALL	All schema-defined fields	All schema-defined fields; new view ID created
ELEMENTS ARE field name	Schema-defined fields named in ELEMENTS clause	Schema-defined fields named in ELEMENTS clause; new view ID created

Note: When the ELEMENTS clause is used for a view ID associated with other subschemas, the subschema compiler ignores the VIEW ID clause, creating a new subschema view with a null ID.

Considerations Specifying Fields in the ELEMENTS Clause

The following considerations apply to copying schema-defined fields into the subschema description of the record:

- Schema-defined fields can be named in any order in the ELEMENTS clause; the order in which they are named is the order in which they will participate in the subschema view of the record.
- If a group field is included in the subschema record description, all of its subordinate fields will be included and will retain their schema-defined order.
- FILLER fields cannot be included in the subschema record description, except as automatically included under groups.
- Redefining fields (or their subordinate fields) cannot be included in the subschema record description. Note, however, that if a redefined field is included, all redefining fields (and their subordinate fields) for that field will be included in the record description.
- Individual fields subordinate to an OCCURS field cannot be included in the record description. The OCCURS field itself must be included, in which case all fields subordinate to it will automatically be included as well.
- If an OCCURS DEPENDING ON field is included, the field on which that fiel d depends must also be included in the record description.
- If an OCCURS DEPENDING ON field is included, it must be named last in the ELEMENTS clause.
- All fields named in the ELEMENTS clause must have the same level number.
- Bit fields cannot be included in the ELEMENTS clause.

PRIORITY Clause Can Optimize Use of Subschema Tables at Runtime

The PRIORITY clause permits the DBA to optimize runtime use of the subschema tables when a frequently used subschema includes many record types, of which only a few are used heavily. Those records used most heavily should be assigned high priorities. The PRIORITY clause is useful primarily in subschemas in which only a few record types are accessed frequently.

Example

This example adds a view of schema record EMPLOYEE to the current subschema. The view includes the employee ID and employee name. Programs accessing the EMPLOYEE record through the current subschema will *not* be able to access other elements defined for the EMPLOYEE record.

```
add record name is employee
   view id is dehview
   elements are emp-id-0415
                emp-name-0415 .
```

SET Statement

The SET statements identify a subschema set. Depending on the verb, the SET statements can also:

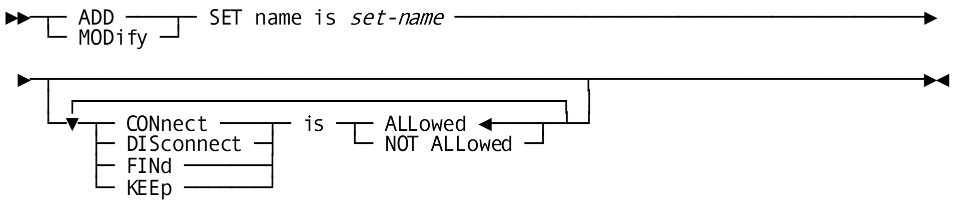
- Copy a set description from the schema
- Determine which DML verbs can be issued against the set
- Delete a set description from the subschema
- Display or punch a subschema set description

The subschema compiler applies SET statements to the current subschema.

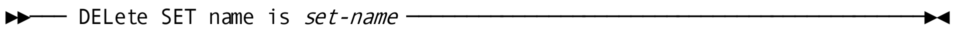
Note: For an explanation of subschema currency, see 9.7, “Establishing Schema and Subschema Currency”.

Syntax

Syntax: ADD/MODIFY SET



Syntax: DELETE SET



Syntax: DISPLAY/PUNCH SET

```
►►─┬─ DISplay ─┬─ SET name is set-name ──────────────────────────────────────►
   └─ PUNCh ───┘

►─┬─────────────────────────────────────────────┬────────────────────────────►
  │  ┌──────────────────────────────────────┐   │
  │  │                    ┌──────────────┐  │   │
  └──▼─┬─ WITh ───────┬───▼─┬─ DETails ─┴──┴───┘
       ├─ ALSo WITh ─┤      ├─ ALL ─────┤
       └─ WITHOut ───┘      └─ NONe ────┘

►─┬──────────────────────────────┬────────────────────────────────────────────►
  └─ VERB ─┬─ ADD ───────────────┘
           ├─ MODify ─┤
           ├─ DELete ─┤
           ├─ DISplay ┤
           └─ PUNCh ──┘

►─┬──────────────────────────┬────────────────────────────────────────────────►
  └─ AS ─┬─ COMments ─┬──────┘
         └─ SYNtax ───┘

►─┬─────────────────────────────────────────┬─────────────────────────────────►◄
  └─ TO ─┬─ module-specification ─┬─────────┘
         └─ SYSpch ───────────────┘
```

Parameters

SET name is *set-name*

Identifies a set defined in the schema associated with the current subschema.

CONnect

Specifies that programs using the current subschema can or cannot issue CONNECT commands against *set-name*.

DISconnect

Specifies that programs using the current subschema can or cannot issue DISCONNECT commands against *set-name*.

FINd

Specifies that programs using the current subschema can or cannot issue FIND commands against *set-name*.

KEEp

Specifies that programs using the current subschema can or cannot issue KEEP commands against *set-name*.

is ALLowed

Specifies that the program using the current subschema can issue the specified DML function against the set. ALLOWED is the default. This clause can be repeated for as many operations as required.

is NOT ALLowed

Specifies that the program using the current subschema cannot issue the specified DML function against the set. This clause can be repeated for as many operations as required.

DETails

Displays and punches access restrictions defined for the set priority defined in the record description.

ALL

Displays and punches the entire set description.

NONe

Displays and punches only the name of the set.

Usage

Effect of ADD on Sets

ADD copies the set description from the schema description into the subschema description.

Before a set can be added to the subschema, the record that owns that set must be present in the subschema. Note, however, that system-owned indexed sets and sets based on native VSAM data sets) are excluded from this rule, since the owner record is not specified in the subschema.

For a set to be a valid subschema component, at least one member record must be present in the subschema.

Note: For information about validation, see 14.8, “VALIDATE Statement".

Effect of MODIFY on Sets

MODIFY modifies some aspect of the set's participation in the subschema. All clauses associated with an ADD operation can be specified for MODIFY operations.

Effect of DELETE on Sets

DELETE removes the set from the current subschema description in the dictionary; the set remains associated with the schema.

Set Automatically Deleted When Owner Record Deleted

If the set's *owner record* is deleted, either from the schema or from the subschema, the set is automatically deleted from the subschema.

Explicitly Delete Set After Deleting Member Records

If the set's *member record* is deleted, either from the schema or from the subschema, the set remains in the subschema. To delete the set, delete all the member records associated with the set before issuing the DELETE SET statement.

Example

In the following example, an attempt is made to add the DEPT-EMPLOYEE set to the current subschema. The subschema compiler returns error messages indicating that the owner of the set (the DEPARTMENT record) has not been added to the subschema:

```
add set name is dept-employee.
```

Produces these messages:

```
*+ E DC643023  OWNER OF SET NOT IN SUBSCHEMA
*+ W DC601017  FORWARD SPACING TO NEXT PERIOD
```

LOGICAL RECORD Statement

The LOGICAL RECORD statements define a logical record that programs using the current subschema can access. Depending on the verb, the LOGICAL RECORD statements can also modify, delete, display, or punch a logical-record description.

A logical record is defined by naming the logical record and all the subschema records that participate in it; these subschema records are known as *logical-record elements*. The records must participate in the subschema (through ADD RECORD statements) before they can be named as logical record elements in the LOGICAL RECORD statement.

Note: IDD work records used as logical-record elements do not need a subschema ADD RECORD statement.

The subschema compiler applies LOGICAL RECORD statements to the current subschema.

Syntax

Syntax: ADD/MODIFY LOGICAL RECORD

```
►►─┬─ ADD ────┬─┬─ LOGical RECord ─┬─ name is logical-record-name ──────►
   └─ MODify ─┘ └─ LR ─────────────┘

 ►─┬─────────────────────────────────────────────────────────┬──────────►
   │                  ┌────────────────────────────────────┐ │
   └─ ELements are ─▼─┬─ subschema-record-specification ─┬─┴─┘
                      └─ idd-record-specification ───────┘

 ►─┬────────────────────────────────┬───────────────────────────────────►
   └─ ON LR-ERROR ─┬─ CLEar ───────┤
                   └─ NOClear ◄────┘

 ►─┬────────────────────────────────────┬───────────────────────────────►
   └─ ON LR-NOT-FOUND ─┬─ CLEar ───────┤
                       └─ NOClear ◄────┘

 ►─┬───────────────────────────────┬────────────────────────────────────►◄
   └─ COMments comment-text ───────┘
```

Expansion of *subschema-record-specification*

```
►►── subschema-record-name ─┬──────────────────────────────┬────────────►◄
                            └─ ROLe name is role-name ─────┘
```

Expansion of *idd-record-specification*

```
►►── idd-record-name ───────────────────────────────────────────────────►

 ►── version-specification ─────────────────────────────────────────────►

 ►─┬───────────────────────────────┬────────────────────────────────────►◄
   └─ ROLe name is role-name ──────┘
```

Syntax: DELETE LOGICAL RECORD

```
►►─── DELete ─┬─ LOGical RECord ─┬─ name is logical-record-name ────────────►◄
              └─ LR ─────────────┘
```

Syntax: DISPLAY/PUNCH LOGICAL RECORD

```
►►─┬─ DISplay ─┬─┬─ LOGical RECord ─┬─ name is logical-record-name ──────────►
   └─ PUNch ───┘ └─ LR ─────────────┘

 ►─┬──────────────────────────────────────────────────┬──────────────────────►
   │ ┌────────────────────────────────────────────┐   │
   │ │                       ┌────────────────┐   │   │
   └─▼─┬─ WITh ──────────┬─▼─┬─ COMments ─┬─┴───┴───┘
       ├─ ALSo WITh ─────┤   ├─ DETails ──┤
       └─ WITHOut ───────┘   ├─ ALL ──────┤
                             └─ NONe ─────┘

 ►─┬─────────────────────────────┬───────────────────────────────────────────►
   └─ VERB ─┬─ ADD ──────────┬───┘
            ├─ MODify ───────┤
            ├─ DELete ───────┤
            ├─ DISplay ──────┤
            └─ PUNch ────────┘

 ►─┬────────────────────────────┬────────────────────────────────────────────►
   └─ AS ─┬─ COMments ─┬────────┘
          └─ SYNtax ───┘

 ►─┬────────────────────────────────────────┬─────────────────────────────────►◄
   └─ TO ─┬─ module-specification ─┬────────┘
          └─ SYSpch ───────────────┘
```

Parameters

LOGical RECord/LR name is *logical-record-name*

Names a logical record. For ADD operations, *logical-record-name* must uniquely identify a logical record in the current subschema.

Logical-record-name cannot duplicate the name of a database record described in the same subschema. Note that synonyms cannot be defined for logical records; logical records with different names (such as names for COBOL versus those for Assembler) must be defined in different subschemas.

Logical-record-name must be a 1- to 16- character name. Note that LOGICAL RECORD and LR are synonymous.

When naming logical records, be sure that the selected names do not conflict with the CA IDMS Data Manipulation Language precompiler with which the logical records will be used.

ELements are

Identifies either a subschema record described in the current subschema or a record described in the dictionary, but not in the schema that owns the current subschema. Multiple subschema and dictionary records can be defined as elements of a logical record.

All elements named in the ELEMENTS clause of the DDL RECORD statement or the DDDL RECORD statement are included in the logical record.

subschema-record-specification

Identifies a record described in the current subschema and optionally assigns a unique ID to a logical record element that occurs more than once in the logical record description.

subschema-record-name

Identifies the name of a subschema record described in the current subschema.

ROLe name is *role-name*

Assigns a unique ID to a logical-record element that occurs more than once in a single logical record; it can also be used for logical-record elements that occur only once in the logical record.

Role-name is a 1- to 16-character name. *Role-name* cannot be the name of a record or record synonym defined in the schema that owns the current subschema, or the name of a logical record used in the subschema.

Each role name can be assigned to only one record type per subschema; it can be assigned to that record type in any number of LOGICAL RECORD statements.

idd-record-specification

Identifies a record described in the dictionary, but cannot be the name of a record or record synonym defined in the schema that owns the current subschema. IDD records commonly are included in logical records to introduce work fields into the logical-record path logic.

idd-record-name

Names the dictionary record.

version-specification

Qualifies the dictionary record with a version number. This clause is required for dictionary records. The version number defaults to the current session option for existing versions.

Note: Expanded syntax for *version-specification* is presented in Chapter 13, “Parameter Expansions".

ROLe name is *role-name*

Assigns a unique ID to a logical-record element that occurs more than once is a single logical record. The syntax rules that appear above apply, with one exception: for dictionary records, *role-name* can be up to 32 characters long.

ON LR-ERROR CLEar

Indicates that variable-storage allocated to the logical record in the program gets set to low values if a program request for access to the named logical record results in the return of the LR-ERROR path status.

ON LR-ERROR NOClear

Indicates that variable-storage allocated to the logical record in the program *does not* get set to low values if a program request for access to the named logical record results in the return of the LR-ERROR path status. NOCLEAR is the default.

ON LR-NOT-FOUND CLEar

Indicates that variable-storage allocated to the logical record in the program gets set to low values if a program request for access to the named logical record results in the return of the LR-NOT-FOUND path status.

ON LR-NOT-FOUND NOClear

Indicates that variable-storage allocated to the logical record in the program *does not* get set to low values if a program request for access to the named logical record results in the return of the LR-NOT-FOUND path status. NOCLEAR is the default.

COMments *comment-text*

Permits documentational entries for the named logical record.

Note: For rules on coding *comment-text*, see 10.5.4, "Coding Comment Text".

COMments

Displays and punches all comment text included in the logical-record definition.

DETails

Displays and punches the following information about the logical record:

- All subschema records that participate as elements in the logical record
- The ON LR-ERROR clause specified for the logical record
- The ON LR-NOT-FOUND clause specified for the logical record

ALL

Displays and punches the entire logical-record description.

NONe

Displays and punches only the name of the logical record.

Usage

Sequence of LR Elements in Program Storage Same as DDL Sequence

When a DML precompiler copies a logical-record description into a program's description of variable storage, each logical-record element is subordinate to the logical record itself. The sequence of logical-record elements in the copied description is the same as that in DDL LOGICAL RECORD statement. If a subschema record occurs more than once in a single logical record, the additional occurrences must be assigned unique IDs, called *roles*.

Must Modify Logical Record if Records Used in Logical Record Change

If any record used as a logical-record element is modified (through the schema compiler or the IDD DDDL compiler), the logical record must also be modified (through the subschema compiler) before the subschema load module is generated.

Must Use Role Names in PATH-GROUP Syntax

Once a role name has been assigned, that name must be used whenever PATH-GROUP syntax requires a logical-record element name.

Document All Logical Record Definitions

The following information should be included as COMMENTS for every logical record defined in the subschema:

- The DML verbs that a program using the subschema can issue in connection with *logical-record-name*
- The selection criteria that a program can include with each permitted logical-record DML verb
- The DBA-defined path statuses that can be returned for each permitted DML verb
- The sequence in which data is returned to the program

Examples

Adding Logical Record Elements

This example adds two subschema records to a newly created logical record:

```
add lr name is dehlr
    elements are employee department.
```

Using Role Names

The following examples compare a valid way to use a role name more than once with an invalid one:

```
VALID

    add lr name is manager-staff
        elements are employee
                     structure
                     employee role name is staff.
    add lr name is dept-roster
        elements are department
                     employee role name is staff.

INVALID

    add lr name is emp-hosp-claims
        elements are employee
                     coverage
                     hospital-claim role name is claim.
    add lr name is emp-dental-claims
        elements are employee
                     coverage
                     dental-claim role name is claim.
```

Note: For more information about logical record path statuses (LR-ERROR and LR-NOT-FOUND), see the *CA IDMS Logical Record Facility Guide.*

PATH-GROUP Statement

The PATH-GROUP statements define, modify, delete, display, or punch processing paths for a specific logical record. At runtime, LRF services program requests by following one of the paths to access the logical record.

For each logical record, at least one path group, and at most four (one for each DML verb that can access the logical record), must be defined. A path group can contain any number of paths. Which path LRF uses at runtime is determined by selection criteria, both in the path group and in the program requesting LRF's services.

The subschema compiler applies PATH-GROUP statements to the current subschema.

Syntax

Syntax: ADD/MODIFY PATH-GROUP

```
►►─┬─ ADD ────┬─ PATh-group name is ─┬─ ERAse ──┬─ logical-record-name ──────►
   └─ MODify ─┘                      ├─ MODify ─┤
                                     ├─ OBTain ─┤
                                     └─ STOre ──┘

 ►─┬──────────────────────────────────────────────────────────────────┬─►◄
   │  ┌─────────────────────────────────────────────────────────────┐ │
   │  │                 ┌─────────────────────────────────────────┐ │ │
   └─▼- select-clause -▼─┬─ compute-clause ─────────────────────┬┴─┴─┘
                        ├─ connect-clause ─────────────────────┤
                        ├─ disconnect-clause ──────────────────┤
                        ├─ erase-clause ───────────────────────┤
                        ├─ evaluate-clause ────────────────────┤
                        ├─ find-obtain-calckey-clause ─────────┤
                        ├─ find-obtain-current-clause ─────────┤
                        ├─ find-obtain-dbkey-clause ───────────┤
                        ├─ find-obtain-index-clause ───────────┤
                        ├─ find-obtain-owner-clause ───────────┤
                        ├─ find-obtain-set-or-area-clause ─────┤
                        ├─ find-obtain-sortkey-clause ─────────┤
                        ├─ get-clause ─────────────────────────┤
                        ├─ if-empty-clause ────────────────────┤
                        ├─ if-member-clause ───────────────────┤
                        ├─ keep-clause ────────────────────────┤
                        ├─ modify-clause ──────────────────────┤
                        ├─ on-error-clause ────────────────────┤
                        └─ store-clause ───────────────────────┘
```

Expansion of *select-clause*

```
►►─── SELect ─────────────────────────────────────────────────────────────►

 ►─┬───────────────────────────────────────┬──────────────────────────────►
   └─ USIng INDex indexed-set-name ────────┘

 ►─┬────────────────────────────────────────────┬─────────────────────────►◄
   └─ for ─┬─ ELement lr-element-name ─────────┤
           ├─ FIEldname lr-field ──────────────┤
           ├─ FIELDNAME-EQ lr-field ───────────┤
           └─ KEYword keyword ─────────────────┘
```

Expansion of *compute-clause*

```
►►─── COMpute lr-field OF LR = ─┬─ 'character-string-literal' ─┬──────────►◄
                                ├─ numeric-literal ────────────┤
                                ├─ arithmetic-expression ──────┤
                                └─ lr-field OF LR ─────────────┘
```

Expansion of *connect-clause*

```
►►─── CONnect database-record-name TO set-name ───────────────────────────►◄
```

Expansion of *disconnect-clause*

```
►►─── DISconnect database-record-name FROM set-name ──────────────────────►◄
```

Expansion of *erase-clause*

```
►►─── ERAse database-record-name ──────────────────────────────────────►

 ►─┬──────────────────────────────────────────────────────────────────►◄
   └─┬─ PERmanent ─┬─ MEMbers ─┘
     ├─ SELECTIVE ─┤
     └─ ALL ───────┘
```

Expansion of *evaluate-clause*

```
►►─┬───────────────────────────────┬──────────────────────────────────►◄
   └─ EVALuate boolean-expression ─┘
```

Expansion of *find-obtain-calckey-clause*

```
►►─┬─ FINd ───┬─┬─────────────────────────┬────────────────────────────►
   └─ OBTain ─┘ └─ KEEp ─┬─────────────┬──┘
                         └─ EXClusive ─┘

 ►─┬─────────────┬─ database-record-name ──────────────────────────────►
   ├─ FIRst ◄ ───┤
   ├─ NEXt ──────┤
   └─ EACh ──────┘

 ►─── WHEre CALckey ─┬─ EQ ─┬─┬─ 'character-string-literal' ─┬─────────►
                     ├─ IS ─┤ ├─ numeric-literal ────────────┤
                     └─ = ──┘ ├─ arithmetic-expression ──────┤
                              └─ lr-field ─┬─ OF LR ──────┬──┘
                                           └─ OF REQUEST ─┘

 ►─┬──────────────────────────┬───────────────────────────────────────►◄
   └─ AND boolean-expression ─┘
```

Expansion of *find-obtain-current-clause*

```
►►─┬─ FINd ───┬─┬─────────────────────────┬────────────────────────────►
   └─ OBTain ─┘ └─ KEEp ─┬─────────────┬──┘
                         └─ EXClusive ─┘

 ►──── CURrent ─┬─ database-record-name ─┬─────────────────────────────►
                ├─ WIThin set-name ──────┤
                └─ WIThin area-name ─────┘

 ►─┬────────────────────────────┬─────────────────────────────────────►◄
   └─ WHEre boolean-expression ─┘
```

Expansion of *find-obtain-dbkey-clause*

```
►►─┬─ FINd ───┬─┬─────────────────────────┬─ database-record-name ─────►
   └─ OBTain ─┘ └─ KEEp ─┬─────────────┬──┘
                         └─ EXClusive ─┘

 ►─── WHEre DBkey ─┬─ EQ ─┬─┬─ numeric-literal ────────────┬───────────►
                   ├─ IS ─┤ ├─ arithmetic-expression ──────┤
                   └─ = ──┘ └─ lr-field ─┬─ OF LR ──────┬──┘
                                         └─ OF REQUEST ─┘

 ►─┬──────────────────────────┬───────────────────────────────────────►◄
   └─ AND boolean-expression ─┘
```

Expansion of *find-obtain-index-clause*

```
►►─┬─ FINd ───┬─┬──────────────────────────┬───────────────────────────────►
   └─ OBTain ─┘ └─ KEEp ─┬────────────────┬┘
                         └─ EXClusive ─┘

►── EACh database-record-name ─────────────────────────────────────────────►

►── USIng ─┬─ INDex ────────────────┬──────────────────────────────────────►
           └─ indexed-set-name ─────┘

►─┬───────────────────────────────┬────────────────────────────────────────►◄
  └─ WHEre boolean-expression ────┘
```

Expansion of *find-obtain-owner-clause*

```
►►─┬─ FINd ───┬─┬──────────────────────────┬───────────────────────────────►
   └─ OBTain ─┘ └─ KEEp ─┬────────────────┬┘
                         └─ EXClusive ─┘

►── OWNer ─┬──────────────────────────┬────────────────────────────────────►
           └─ database-record-name ───┘

►── WIThin set-name ───────────────────────────────────────────────────────►

►─┬───────────────────────────────┬────────────────────────────────────────►◄
  └─ WHEre boolean-expression ────┘
```

Expansion of *find-obtain-set-or-area-clause*

```
►►─┬─ FINd ───┬─┬──────────────────────────┬───────────────────────────────►
   └─ OBTain ─┘ └─ KEEp ─┬────────────────┬┘
                         └─ EXClusive ─┘

►─┬─ FIRst ───────┬─ database-record-name ─────────────────────────────────►
  ├─ LASt ────────┤
  ├─ NEXt ────────┤
  ├─ PRIor ───────┤
  ├─ EACh ────────┤
  └─ EACh PRIor ──┘

►── WIThin ─┬─ set-name ──┬────────────────────────────────────────────────►
            └─ area-name ─┘

►─┬───────────────────────────────┬────────────────────────────────────────►◄
  └─ WHEre boolean-expression ────┘
```

Expansion of *find-obtain-sortkey-clause*

```
►►─┬─ FINd ───┬─┬──────────────────────────┬───────────────────────────────►
   └─ OBTain ─┘ └─ KEEp ─┬────────────────┬┘
                         └─ EXClusive ─┘

►─┬──────────────┬─ database-record-name ──────────────────────────────────►
  ├─ FIRst ◄─────┤
  └─ EACh ───────┘

►── WIThin set-name ───────────────────────────────────────────────────────►

►── WHEre SORtkey ─┬─ EQ ─┬─┬─ 'character-string-literal' ───────────┬─────►
                   ├─ IS ─┤ ├─ numeric-literal ──────────────────────┤
                   └─ = ──┘ ├─ arithmetic-expression ────────────────┤
                            └─ lr-field ─┬─ OF LR ──────┬────────────┘
                                         └─ OF REQUEST ─┘

►─┬────────────────────────────┬───────────────────────────────────────────►◄
  └─ AND boolean-expression ───┘
```

Expansion of *get-clause*

```
►►── GET database-record-name ─────────────────────────────────────────►◄
```

Expansion of *if-empty-clause*

```
►►── IF set-name is ─┬───────┬─ EMPty ────────────────────────────────►◄
                     └─ NOT ─┘
```

Expansion of *if-member-clause*

```
►►── IF ─┬───────┬─ set-name MEMber ──────────────────────────────────►◄
         └─ NOT ─┘
```

Expansion of *keep-clause*

```
►►── KEEp ─┬─────────────┬─ CURrent ─┬─ database-record-name ─┬───────►◄
           └─ EXClusive ─┘           ├─ WIThin set-name ──────┤
                                     └─ WIThin area-name ─────┘
```

Expansion of *modify-clause*

```
►►── MODify database-record-name ─────────────────────────────────────►◄
```

Expansion of *on-error-clause*

```
►►── ON idms-error-status ─┬─ DO nested-block END ─────────┬──────────►◄
                           ├─ ITErate ─────────────────────┤
                           ├─ NEXt ────────────────────────┤
                           └─┬─────────┬─ RETurn path-status ─┘
                             └─ CLEar ─┘
```

Expansion of *store-clause*

```
►►── STOre database-record-name ──────────────────────────────────────►◄
```

Syntax: DELETE PATH-GROUP

```
►►── DELete PATh-group name is ─┬─ ERAse ──┬─ logical-record-name ────►◄
                                ├─ MODify ─┤
                                ├─ OBTain ─┤
                                └─ STOre ──┘
```

Syntax: DISPLAY/PUNCH PATH-GROUP

```
►►─┬─ DISplay ─┬─ PATh-group-name is ─┬─ ERAse ──┬─ logical-record-name ──────►
   └─ PUNch ───┘                      ├─ MODify ─┤
                                      ├─ OBTain ─┤
                                      └─ STOre ──┘

►─┬────────────────────────────────────────────┬──────────────────────────►
  │ ┌────────────────────────────────────────┐ │
  │ │                  ┌───────────────────┐ │ │
  └─▼─┬─ WITh ───────┬─▼─┬─ DETails ─┬─────┴─┴─┘
      ├─ ALSo WITh ──┤   ├─ ALL ─────┤
      └─ WITHOut ────┘   └─ NONe ────┘

►─┬───────────────────────────┬───────────────────────────────────────────►
  └─ VERB ─┬─ ADD ───────┬────┘
           ├─ MODify ────┤
           ├─ DELete ────┤
           ├─ DISplay ───┤
           └─ PUNch ─────┘

►─┬──────────────────────────┬────────────────────────────────────────────►
  └─ AS ─┬─ COMments ─┬──────┘
         └─ SYNtax ───┘

►─┬─────────────────────────────────────┬─────────────────────────────────►◄
  └─ TO ─┬─ module-specification ─┬─────┘
         └─ SYSpch ───────────────┘
```

Parameters

PATh-group name is ERAse *logical-record-name*

Specifies an ERASE path-group for which the subsequent path definitions are available to service program requests. *Logical-record-name* must be the name of a logical record defined for the current subschema.

PATh-group name is MODify *logical-record-name*

Specifies a MODIFY path-group for which the subsequent path definitions are available to service program requests. *Logical-record-name* must be the name of a logical record defined for the current subschema.

PATh-group name is OBTain *logical-record-name*

Specifies an OBTAIN path-group for which the subsequent path definitions are available to service program requests. *Logical-record-name* must be the name of a logical record defined for the current subschema.

PATh-group name is STOre *logical-record-name*

Specifies a STORE path-group for which the subsequent path definitions are available to service program requests. *Logical-record-name* must be the name of a logical record defined for the current subschema.

select-clause

Delimits paths within a path group. Thus, at least one SELECT clause must precede the database commands that constitute a path definition.

Multiple paths can be defined for a single path group; LRF executes only one path per program request. LRF chooses that path based on the *selectors* coded in the FOR options of the multiple SELECT clauses. The first SELECT clause whose selectors match those of the program request is the path that LRF executes.

USIng INDex *indexed-set-name*

Identifies the indexed set (if any) that LRF uses when executing a database command specified using the *find-obtain-index-clause* (described later). *Indexed-set-name* must be a sorted indexed set included in the current subschema. When coded, the USING INDEX clause must precede the FOR clause(s) of the SELECT clause.

for

Identifies selectors to be used as the basis of path selection to service logical-record requests. For a path to be chosen, the WHERE clause of the program DML request must supply information that matches all selectors specified in any one of the path's SELECT clauses.

A SELECT clause can contain any number of selectors, including zero. A SELECT clause with no selectors will always cause the path to be selected. Four types of selectors can be included in the SELECT clause, in any combination: KEYWORD, FIELDNAME-EQ, FIELDNAME, and ELEMENT.

ELement *lr-element-name*

Specifies that the WHERE clause of a request to be serviced by the path must reference a field in the named logical-record element (database record) in any manner.

FIEldname *lr-field*

Specifies that a request to be serviced by the path must reference the named logical-record field (in any manner).

The optional qualifier OF *lr-element-name* names the logical-record element that contains the logical-record field. This qualifier is required if *lr-field-name* is not unique within the subschema.

Note: Expanded syntax for *lr-field* is presented in Chapter 13, “Parameter Expansions”.

FIELDNAME-EQ ***lr-field***

Specifies that the WHERE clause of a request to be serviced by the path must reference the named logical-record field in a logically conjunctive single-value equality comparison. For example, LRF will service the following requests:

```
where fieldname eq 123
where fieldname eq 123 and ...
```

The following requests will not be serviced:

```
where lr-field-name eq 12 / 3
where lr-field-name eq 123 or ...
```

The optional qualifier OF *lr-element-name* names the logical-record element that contains the logical-record field. This qualifier is required if *lr-field-name* is not unique within the subschema. FIELDNAME-EQ selectors are intended for paths that utilize CALCKEY, SORTKEY, or DBKEY access. Therefore, the named field is usually qualified with an OF REQUEST clause in a path DML statement, but not in the SELECT statement

Note: Expanded syntax for *lr-field* is presented in Chapter 13, “Parameter Expansions”.

KEYword ***keyword***

Specifies that the WHERE clause of a request to be serviced by the path must include the named keyword in an affirmative and logically conjunctive manner. For example, LRF will service the following types of requests:

```
where keyword
where keyword and ...
```

The following types of requests will not be serviced:

```
where not keyword
where keyword or ...
```

compute-clause

Sets the value of the left operand (*lr-field-name*) to equal the value represented by the right operand. Note that all named fields used with COMPUTE must be fields within the logical record named in the PATH-GROUP statement.

***lr-field* OF LR**

In the left operand, *lr-field* names the receiving data field. If *logical-record-field-name* occurs more than once within the logical record, it must be qualified by OF *lr-element-name*. *Lr-element-name* must identify the logical-record element containing the data field, as follows:

- If the logical-record element was assigned a role name in the LOGICAL RECORD statement, *lr-element-name* must specify that role name.
- If the logical-record element was not assigned a role name in the LOGICAL RECORD statement, *lr-element-name* must specify the logical-record element name.

Note: Expanded syntax for *lr-field* is presented in Chapter 13, “Parameter Expansions”.

'*character-string-literal*'

Specifies an alphanumeric literal enclosed in single quotes.

numeric-literal

Specifies a numeric literal as the right operand. A minus sign (-) can precede the numeric literal.

arithmetic-expression

Specifies either a simple arithmetic expression (containing only 1 operator) or a compound arithmetic expression (containing multiple operators). Arithmetic operators permitted in an arithmetic expression are +, -, *, and /. Operands can be numeric literals (without quotes) and logical-record field names.

***lr-field* OF LR**

In the right operand, specifies a data field that participates in the current logical record. Rules for qualifying this name are the same as those for qualifying the left operand.

connect-clause

Establishes the current occurrence of the named database record as a member of the current occurrence of the named set.

database-record-name

Names the type of record to be connected. *Database-record-name* must be included in the current subschema.

set-name

Names the set to which the database record will be connected. *Set-name* must be included in the current subschema.

disconnect-clause

Disconnects the current occurrence of the named database record from the current occurrence of the named set.

database-record-name

Names the type of record to be disconnected. *Database-record-name* must be included in the current subschema.

set-name

Names the set from which the database record will be disconnected. *Set-name* must be included in the current subschema.

erase-clause

Erases the current occurrence of the named database record.

database-record-name

Specifies the type of record to be erased. *Database-record-name* must be included in the current subschema.

PERmanent MEMbers

Erases the specified record and its mandatory set members. Optional member records are disconnected but not erased. All erased mandatory members that, in turn, own set occurrences are treated as if ERASE PERMANENT commands had been issued for those erased records (that is, all mandatory members of the erased records' sets are also erased). This process continues through the database structure until all mandatory records in the sequence have been treated.

SELECTIVE MEMbers

Erases the specified record and its mandatory set members. Optional member records are erased only if they do not currently participate as members in other set occurrences. All erased records that, in turn, own set occurrences are treated as if ERASE SELECTIVE commands had been issued for those erased records.

ALL MEMbers

Erases the specified record and all of its mandatory and optional set members. All erased records that, in turn, own set occurrences are treated as if ERASE ALL commands had been issued for those erased records.

evaluate-clause

Determines whether the specified boolean expression is true or false, allowing specific PATH-GROUP logic to be performed based on the outcome of the evaluation.

If the expression is true, CA IDMS/DB returns an error status of 0000. If the expression is false, CA IDMS/DB returns an error status of 2001. The error status can be checked with the ON clause, thus allowing conditional processing. Use of EVALUATE implies ON 0000 NEXT and ON 2001 ITERATE.

boolean-expression

Specifies a boolean expression. In EVALUATE, comparisons within the boolean expression must specify *logical-record-field-name* OF LR.

Note: Expanded syntax for *boolean-expression* is presented in Chapter 13, “Parameter Expansions”.

find-obtain-calckey-clause

Specifies that a database record is to be located or obtained by means of its CALC key.

FINd

Finds (locates) the named database record.

OBTain

Finds (locates) and obtains the named database record.

KEEp

Places a shared lock on the record occurrence.

EXClusive

Places an exclusive lock on the record occurrence.

FIRst

Specifies that the first record occurrence encountered containing the indicated CALC-key value is to be accessed. FIRST is the default.

NEXt

Specifies that a record occurrence containing the same CALC-key value as the current record of the specified record type is to be accessed. NEXT assumes previous retrieval of a record containing the specified CALC-key value and accesses a record containing a duplicate CALC key.

EACh

Specifies that each record containing the indicated CALC-key value is to be accessed. EACH indicates that this FIND/OBTAIN command can be iterated.

Every time the command is iterated, LRF retrieves another occurrence of the named record that contains the specified CALC key. This iteration permits LRF to access all records that contain that CALC key.

database-record-name

Specifies the type of record to be accessed. *Database-record-name* must be a record whose location mode is CALC.

WHEre CALCkey EQ/IS/=

Specifies the CALC-key value to be used when accessing the database record.

'*character-string-literal*'

Specifies an alphanumeric literal enclosed in single quotes.

numeric-literal

Specifies a numeric value to be used as the CALC key.

arithmetic-expression

Specifies an arithmetic expression whose result is to be used as the CALC key. The expression can be designated as a simple arithmetic operation or as a compound arithmetic operation. Arithmetic operators permitted in an arithmetic expression are +, -, *, and /. Operands can be literals, logical-record fields, or database fields.

lr-field

Specifies that the CALC-key value to be used is in the named logical-record field. If the database record's CALC key is made up of noncontiguous fields, *logical-record-field-name* must be the same size as the total length of all fields in the CALC key. To accomplish this, define an IDD record type that contains *logical-record-field-name* and name the IDD record as an element of the logical record.

Note: Expanded syntax for *lr-field* is presented in Chapter 13, “Parameter Expansions”.

OF LR

Specifies that the CALC-key value to be used is in the named logical-record field in program variable storage. The path DML statement must initialize the field to the appropriate value before the FIND/OBTAIN request is issued. Note that LRF uses the contents of the named field, even if the request's WHERE clause also specifies a CALC-key value.

OF REQUEST

Specifies that the CALC-key value is passed in the request's WHERE clause. *Logical-record-field-name* is equated in the WHERE clause to a literal value, a program variable, or the value of a logical-record field. Note that if OF REQUEST is specified, *logical-record-field-name* should also be named in a SELECT FOR FIELDNAME-EQ clause in the path containing this FIND/OBTAIN command.

AND *boolean-expression*

Specifies boolean selection criteria that further identify the database record occurrence to be accessed.

Note: Expanded syntax for *boolean-expression* is presented in Chapter 13, “Parameter Expansions”.

find-obtain-current-clause

Specifies that the database record that is current of the named record type, set, or area is to be located or obtained.

FINd

Finds (locates) the named database record.

OBTain

Finds (locates) and obtains the named database record.

KEEp

Places a shared lock on the record occurrence.

EXClusive

Places an exclusive lock on the record occurrence.

database-record-name

Specifies the type of record to be accessed. *Database-record-name* must be included in the current subschema.

WIThin *set-name*

Specifies the database record occurrence that is current of the named set. *Set-name* must be included in the current subschema.

WIThin *area-name*

Specifies the database record occurrence that is current of the named area. *Area-name* must be included in the current subschema.

WHEre *boolean-expression*

Specifies boolean selection criteria that further identify the database record occurrence to be accessed.

Note: Expanded syntax for *boolean-expression* is presented in Chapter 13, "Parameter Expansions".

find-obtain-dbkey-clause

Specifies that a database record is to be located or obtained by means of its db-key.

FINd

Finds (locates) the named database record.

OBTain

Finds (locates) and obtains the named database record.

KEEp

Places a shared lock on the record occurrence.

EXClusive

Places an exclusive lock on the record occurrence.

database-record-name

Specifies the type of record to be accessed.

number-literal

Specifies a literal value to be used as the db-key. *Numeric-literal* must be a 1- to 10-digit unsigned numeric value.

arithmetic-expression

Specifies an arithmetic expression whose result is to be used as the db-key. The expression can be designated as a simple arithmetic operation or as a compound arithmetic operation. Arithmetic operators permitted in an arithmetic expression are +, -, *, and /. Operands can be a literal, logical-record field, and database field.

lr-field

Specifies that the value in the named field is to be used as the db-key. *Logical-record-field-name* must be a full binary field or a 4-byte packed (COMP-3) field.

Note: Expanded syntax for *lr-field* is presented in Chapter 13, “Parameter Expansions”.

OF LR

Specifies that the db-key value to be used is in the named logical-record field in program variable storage. The path DML statement must initialize the field to the appropriate value before the FIND/OBTAIN request is issued. The value of the named field is a fullword binary value; if the field is a packed data field, CA IDMS/DB converts its value to binary. Note that LRF uses the contents of the named field, even if the request's WHERE clause also specifies a db-key value.

OF REQUEST

Specifies that the db-key value is passed in the request's WHERE clause. *Logical-record-field-name* is equated in the WHERE clause to a literal value, to a fullword binary field, or to a logical-record field that contains a fullword binary value (if the field is a packed data field, CA IDMS/DB converts its value to binary). Note that if OF REQUEST is specified, *logical-record-field-name* should also be named in a SELECT FOR FIELDNAME-EQ clause in the path containing this FIND/OBTAIN command.

AND *boolean-expression*

Specifies boolean selection criteria that further identify the database record occurrence to be accessed.

Note: Expanded syntax for *boolean-expression* is presented in Chapter 13, “Parameter Expansions”.

find-obtain-index-clause

Specifies that a database record is to be located or obtained on the basis of its membership within a sorted indexed set.

FINd

Finds (locates) the named database record. In this clause, FIND searches the index and thus does not establish currency for *database-record-name*. To establish such currency, use OBTAIN.

OBTain

Finds (locates) and obtains the named database record.

KEEp

Places a shared lock on the record occurrence.

EXClusive

Places an exclusive lock on the record occurrence.

EACh *database-record-name*

Specifies that each member of the indexed set is to be accessed.

EACH specifies that each member of the indexed set is to be accessed. EACH indicates that this FIND/OBTAIN command can be iterated. Every time the command is iterated, LRF retrieves another occurrence of the named record via the index. This iteration permits LRF to access all records in the indexed set.

Database-record-name specifies the type of record to be accessed. It must name a record defined as a member of the named set.

USIng INDex

Indicates that the set used for retrieval is the set named in the USING INDEX clause of the SELECT clause that caused the path to be selected. That is, the set name coded in the SELECT clause replaces the word INDEX when LRF interprets the DML command.

In the following example, a program request that includes a reference to EMP-NAME causes LRF to interpret the path DML command as OBTAIN EACH EMPLOYEE USING IND-EMP-NAME. A program request that includes a reference to EMP-ZIP-CODE causes LRF to interpret the command as OBTAIN EACH EMPLOYEE USING IND-EMP-ZIP-CODE.

```
add path-group name is obtain lr-employee
    select using index ind-emp-name
           for fieldname emp-name
    select using index ind-emp-zip-code
           for fieldname emp-zip-code
    obtain each employee using index.
```

USIng ***indexed-set-name***

Identifies the name of a sorted indexed set to which *database-record-name* belongs. *Indexed-set-name* must be included in the current subschema. This option must be used if the path's SELECT clause does not include USING INDEX for the set.

WHEre ***boolean-expression***

Specifies boolean selection criteria that further identify the database record occurrence to be accessed. If the WHERE clause contains any reference to the indexed set's sort control element, LRF uses the index (rather than checking values in each record) to satisfy the WHERE clause criteria.

Note: Expanded syntax for *boolean-expression* is presented in Chapter 13, “Parameter Expansions”.

find-obtain-owner-clause

Specifies that the owner of the current occurrence of the named set is to be located or obtained.

FINd

Finds (locates) the named database record.

OBTain

Finds (locates) and obtains the named database record.

KEEp

Places a shared lock on the record occurrence.

EXClusive

Places an exclusive lock on the record occurrence.

OWNer ***database-record-name***

Identifies the occurrence (role) of the set's owner as a record within the subschema. *Database-record-name* need not be coded if the owner record is specified only once in the LOGICAL RECORD statement.

WIThin ***set-name***

Specifies the set owned by the database record. *Set-name* must be included in the current subschema.

Note: If the set membership option for the named set is not mandatory automatic, the path should test for set membership before issuing this command.

WHEre ***boolean-expression***

Specifies boolean selection criteria that further identify the database record occurrence to be accessed.

Note: Expanded syntax for *boolean-expression* is presented in Chapter 13, “Parameter Expansions”.

find-obtain-set-or-area-clause

Specifies that a database record is to be located or obtained on the basis of its logical location with a set or its physical location within an area.

FINd

Finds (locates) the named database record.

OBTain

Finds (locates) and obtains the named database record.

KEEp

Places a shared lock on the record occurrence.

EXClusive

Places an exclusive lock on the record occurrence.

FIRst

Specifies that the first record in the named set or area is to be accessed.

LASt

Specifies that the last record in the named set or area is to be accessed.

NEXt

Specifies that the next record in the named set or area is to be accessed. NEXT assumes that currency has been established in the named set or area and accesses the next record in relation to the record previously accessed in the set or area by either program request or path command.

PRIor

Specifies that the prior record in the named set or area is to be accessed. PRIOR assumes that currency has been established in the named set or area and accesses the prior record in relation to the record previously accessed in the set or area by either program request or path command.

EACh

Specifies that each record in the named set or area is to be accessed, beginning with the first record occurrence in the set or area. EACH indicates that this FIND/OBTAIN command can be iterated.

Each time the command is iterated, the next record occurrence is accessed in the set or area, based on the currency established by the previous execution of the command. This iteration permits LRF to walk the named set or sweep the named area.

EACH PRIor

Specifies that each prior record occurrence in the set or area is to be accessed, beginning with the last record occurrence in the set or area. EACH PRIOR indicates that this FIND/OBTAIN command can be iterated.

Each time the command is iterated, the prior record occurrence in the set or area is accessed, based on the currency established by the previous execution of the command. This iteration permits LRF to walk the named set or sweep the named area in a prior direction.

database-record-name

Specifies the type of record to be accessed. *Database-record-name* must be included in the current subschema.

WIThin *set-name*

Specifies a database record occurrence that is defined to the named set. *Set-name* must be included in the current subschema.

WIThin *area-name*

Specifies a database record occurrence that is defined to the named area. *Area-name* must be included in the current subschema.

WHEre *boolean-expression*

Specifies boolean selection criteria that further identify the database record occurrence to be accessed.

Note: Expanded syntax for *boolean-expression* is presented in Chapter 13, “Parameter Expansions”.

find-obtain-sortkey-clause

Specifies that a database record in a sorted set is to be accessed on the basis of its sort-key value.

FINd

Finds (locates) the named database record.

OBTain

Finds (locates) and obtains the named database record.

KEEp

Places a shared lock on the record occurrence.

EXClusive

Places an exclusive lock on the record occurrence.

FIRst

Specifies that the first record occurrence encountered containing the indicated sort-key value is to be accessed. FIRST is the default.

EACh

Specifies that each record containing the indicated sort-key value is to be accessed. EACH indicates that this FIND/OBTAIN command can be iterated.

Every time the command is iterated, LRF retrieves another occurrence of the named record that contains the specified sort key. This iteration permits LRF to access all records that contain that sort key.

database-record-name

Specifies the type of record to be accessed. *Database-record-name* must be defined as a member of the named set.

WIThin *set-name*

Specifies the set of which *database-record-name* is a member. *Set-name* must be included in the current subschema.

WHEre SORtkey EQ/IS/=

Specifies the sort-key value to be used when accessing the database record.

'*character-string-literal*'

Specifies an alphanumeric literal enclosed in single quotes.

numeric-literal

Specifies a numeric value to be used as the sort key.

arithmetic-expression

Specifies an arithmetic expression whose result is to be used as the sort key. The expression can be designated as a simple arithmetic operation or as a compound arithmetic operation. Arithmetic operators permitted in an arithmetic expression are +, -, *, and /. Operands can be literals, logical-record fields, or database fields.

lr-field

Specifies that the sort-key value to be used is in the named logical-record field. If the sort key is made up of noncontiguous fields, *logical-record-field-name* must be the same size as the total length of all fields in the sort key. To accomplish this, define an IDD record type that contains *logical-record-field-name* and name the IDD record as an element of the logical record.

Note: Expanded syntax for *lr-field* is presented in Chapter 13, “Parameter Expansions”.

OF LR

Specifies that the sort-key value to be used is in the named logical-record field in program variable storage. The path DML statement must initialize the field to the appropriate value before the FIND/OBTAIN request is issued. Note that LRF uses the contents of the named field, even if the request's WHERE clause also specifies a sort-key value.

OF REQUEST

Specifies that the sort-key value is passed in the request's WHERE clause. *Logical-record-field-name* is equated in the WHERE clause to a literal value, a program variable, or the value of a logical-record field. Note that if OF REQUEST is specified, *logical-record-field-name* should also be named in a SELECT FOR FIELDNAME-EQ clause in the path containing this FIND/OBTAIN command.

AND *boolean-expression*

Specifies boolean selection criteria that further identify the database record occurrence to be accessed.

Note: Expanded syntax for *boolean-expression* is presented in Chapter 13, “Parameter Expansions”.

get-clause

Moves the located occurrence of the named database record to the corresponding logical-record element in the variable-storage location assigned to the logical record named in the PATH-GROUP NAME clause.

database-record-name

Specifies the type of record to be moved. *Database-record-name* must be included as a logical-record element.

if-empty-clause

Tests the current occurrence of the named set to determine whether it contains any member record occurrences. If the set does not contain members, the error status is set to 0000; otherwise the error status is set to 1601.

set-name

Specifies the name of the set to be tested. *Set-name* must be included in the current subschema.

NOT

Reverses the default ON conditions for the IF SET EMPTY command. Refer to "Usage" for specific path commands using default ON clauses.

if-member-clause

Tests the record that is current of run unit to determine whether it participates as a member of any occurrence of the named set.

If the record *is* a member of the set, the error status is set to 0000; otherwise, the error status is set to 1601. Refer to "Usage" for specific path commands using default ON clauses.

NOT

Reverses the default ON conditions for the IF SET MEMBER command.

set-name

Names the set on which the member test is to be performed. Evaluates to true if the current record of run unit *does* participate as a member of any occurrence of the named set. *Set-name* must be included in the current subschema.

keep-clause

Places a shared or exclusive lock on the record occurrence that is current of the named record type, set, or area.

KEEp CURrent

Places a shared lock or the current record occurrence.

KEEp EXClusive CURrent

Places an exclusive lock or the current record occurrence.

database-record-name

Places the lock on the current occurrence of the named database record type. *Database-record-name* must be included in the current subschema.

WIThin *set-name*

Places the lock on the current occurrence of the named set. *Set-name* must be included in the current subschema.

WIThin *area-name*

Places the lock on the current occurrence of the named area. *Area-name* must be included in the current subschema.

modify-clause

Modifies the named database record by using data present in the variable-storage location assigned to the logical record. The requesting program must initialize variable storage to the appropriate value before LRF executes this path command.

database-record-name

Identifies a logical-record element of the logical record named in the PATH-GROUP statement.

on-error-clause

Specifies the action to be taken in the event that CA IDMS/DB returns the error status indicated by *idms-error-status*. This path command can be used to override the default ON clauses generated automatically by the subschema compiler as shown in the "Usage" topic.

idms-error-status

Specifies a 4-digit value of which the first two digits represent the CA IDMS/DB major error code and the last two digits represent the minor error code value.

Note: For more information about CA IDMS/DB runtime error-status codes, see the *CA IDMS Navigational DML Programming Guide*.

DO *nested-block* END

Specifies that a nested block of path commands following this ON command is to be executed. The keyword END is required at the termination of the nested block. The block of commands included with an ON DO command can itself include ON DO statements; up to 32 levels of nested blocks are permitted.

ITErate

Specifies that the most recent successfully executed path command containing an EACH clause is to be reexecuted. See the table under "Usage" in this section for a list of the ON ITERATE clauses generated automatically by the subschema compiler.

NEXt

Specifies that if CA IDMS/DB returns *idms-error-status*, the next command in the path is to be executed. The subschema compiler automatically generates an ON 0000 NEXT command for every path command (with the exception of IF NOT EMPTY and IF NOT MEMBER, for which it generates ON 1601 NEXT).

RETurn *path-status*

Specifies that LRF is to interrupt path processing and return *path-status* to the requesting program. (An LR-NOT-FOUND path status *terminates* path processing.) *Path-status* must be a 1- to 16-character alphanumeric string, without enclosing quotes.

CLEar RETurn *path-status*

Specifies that the contents of the logical record in program variable storage are to be cleared to low values. If CLEAR is not specified, LRF will return partial logical records.

store-clause

Stores a new occurrence of the named database record by using data present in the variable-storage location assigned to the logical record. The requesting program must initialize variable storage to the appropriate value before LRF executes this path command.

database-record-name

Identifies a logical-record element of the logical record named in the PATH-GROUP statement.

DETails

Display and punches the entire path group description.

ALL

Display and punches the entire path group description.

NONe

Display and punches only the name of the path group.

Usage

Path Group Can Include Multiple Paths

A path group can include any number of paths, each of which must be preceded by at least one SELECT clause.

Commands Allowed for OBTAIN Path-Groups

Paths included in a path group for OBTAIN logical-record requests cannot include database modification commands (MODIFY, STORE, ERASE, CONNECT, DISCONNECT). OBTAIN paths can include only the following database commands:

- FIND (all formats)
- OBTAIN (all formats)
- GET
- KEEP

Identifying a Database Record in a Path-Group

The path DML commands require identification of the database record to be acted upon. Specify the database record as follows:

- If the database record is not an element of the path group's logical record, specify the database record name as defined in the subschema.
- If the database record is an element of the path group's logical record, specify one of the following names:
  - Logical-record element name—If the database record's position within the logical record is not assigned a role name, specify the logical-record element name (the subschema record name).
  - Role name—If the database record's position within the logical record is assigned a role name, specify the role name.

    Note: For more information about the position of logical record elements within logical records, see 15.6, "LOGICAL RECORD Statement".

Access Restrictions Apply to Logical Record Navigation

Access restrictions specified for records, sets, and areas apply to DML commands included in paths.

Terminate PATH-GROUP Statement With a Period

Each PATH-GROUP statement contains only one period, at the very end of the statement.

Logical Record Placed in Program Variable Storage

For OBTAIN and GET path DML commands, the retrieved record is placed in the program's variable storage. The record is placed in its corresponding logical-record element within the logical record named in the PATH GROUP statement. For MODIFY and STORE path DML statements, the data used to update the database is taken from this same location.

Note: In a DML program, the programmer can specify that the record be placed in and taken from an alternative variable-storage location. To do this, the programmer codes an INTO clause on the OBTAIN logical-record request and a FROM clause on the MODIFY, STORE, and ERASE logical-record requests. See the *CA IDMS Navigational DML Programming Guide* for details.

Coding find-obtain-index-clause can reduce I/O

When LRF encounters the *find-obtain-index-clause*, it looks for any reference to the set's sort control element in the WHERE clauses of both this path's command and the program request. If such a reference is found, LRF uses the index (rather than checking values in each record) to satisfy the WHERE clause criteria. Using the index usually takes fewer I/O operations than does checking each member record's sort-key value. Thus, when accessing each member of a sorted indexed set, this form of FIND/OBTAIN is preferable to another.

Default ON Clauses for Specific Path Commands

This table shows the path commands for which ON clauses are automatically generated by the subschema compiler. These ON clauses are overridden by the path definition.

Path Command	Default ON Clause
FIND/OBTAIN WHERE DBKEY	ON 0000 NEXT
FIND/OBTAIN WHERE CALCKEY	ON 0326 ITERATE
FIND/OBTAIN WITHIN SET WHERE SORTKEY	
FIND/OBTAIN WITHIN SET USING INDEX	
FIND/OBTAIN WITHIN SET/AREA	ON 0000 NEXT
	ON 0307 ITERATE
IF SET EMPTY	ON 0000 NEXT
IF SET MEMBER	ON 1601 ITERATE
IF NOT SET EMPTY	ON 0000 ITERATE
IF NOT SET MEMBER	ON 1601 NEXT
FIND/OBTAIN CURRENT	ON 0000 NEXT
FIND/OBTAIN OWNER WITHIN SET	
GET	
MODIFY	
STORE	
CONNECT	
DISCONNECT	
ERASE	
KEEP	
COMPUTE	
EVALUATE	ON 0000 NEXT
	ON 2001 ITERATE

On-error-clause follows if-empty/member-clause

An *on-error-clause* follows a *if-member-clause* or a *if-empty-clause* either explicitly, if coded by the DBA, or implicitly by the subschema compiler. The *on-error-clause* indicates the action to be taken based on the error-status code returned by CA IDMS/DB.

Example

This example modifies the OBTAIN DEHLR path group. The SELECT statement of the path group obtains employee records using the employee ID as the CALC key:

```
mod path-group name is obtain dehlr
    select for element employee
    obtain employee where calckey is emp-id-0415 of request.
```

Note: For more information about logical records and path groups, see the *CA IDMS Logical Record Facility Guide*.

VALIDATE Statement

The VALIDATE statement instructs the subschema compiler to verify the relationships among all components of the subschema that is current for update. If no errors occur during the validation, the subschema compiler sets the status of the subschema to VALID; if errors exist, the subschema compiler sets the status to IN ERROR.

Authorization

The user requires authorization to modify the current subschema.

Note: See the USER clause under the SUBSCHEMA statements for more information.

Syntax

```
▶▶─── VALIDATE ──────────────────────────────────────────────────▶◀
```

Usage

Effect of VALIDATE on Subschemas

When the subschema compiler validates the subschema, it takes one of the following actions:

- If it finds no errors, the compiler sets the subschema's status to VALID. A VALID status means the subschema load module can be generated.
- If it finds errors, the compiler issues messages indicating the exact nature of each error and sets the subschema's status to IN ERROR. The DBA uses these messages to determine what changes must be made for the subschema to be valid.

Must Validate the Subschema Following ADD and MODIFY

The subschema compiler also sets the subschema's status to IN ERROR under these conditions:

- The subschema was just created with an ADD SUBSCHEMA statement
- The subschema was modified with a MODIFY SUBSCHEMA statement
- The schema associated with the subschema was modified in a way that affects the subschema; for example, a set deletion
- Any component of the subschema as added, modified, or deleted

VALIDATE Typically Used to Check Errors

VALIDATE is typically used for dry runs of the subschema compiler, since it causes the compiler to check the components but not to create subschema load modules.

GENERATE Statement

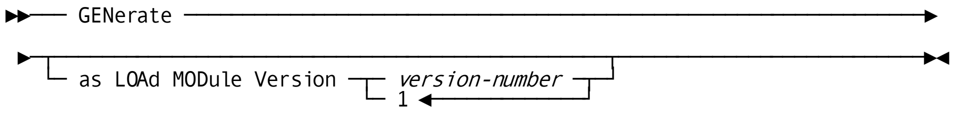
The GENERATE statement instructs the compiler to create subschema tables for the subschema that is current for update and to store them as a load module in the dictionary load area. For GENERATE to produce the new subschema load module, the current subschema must be valid. So, if a VALIDATE statement has not been specified for the subschema, the GENERATE statement causes the compiler to perform validation before creating the subschema tables.

Authorization

The user requires authorization to modify the current subschema.

Note: See the USER clause under the SUBSCHEMA statements for more information.

Syntax



Parameters

as LOAd MODule Version *version-number*

Specifies the version number to be assigned to the subschema load module. *Version-number* must be an unsigned integer in the range 1 through 9999. 1 is the default.

Note: Unlike other version numbers, the load module version number does not default to the current session option.

LOAD MODULE Statement

The LOAD MODULE statements identify a subschema load module stored in the load area of the dictionary (DDLDCLOD). The MODIFY and DELETE statements update the load module stored in the dictionary load area.

A load module is stored in the dictionary load area as a result of one of the following statements:

- A subschema GENERATE statement
- An IDD DDDL ADD LOAD MODULE statement followed by an object deck

Only a load module identified with a load module type of SUBSCHEMA can be processed.

Depending on the verb and options submitted to the subschema compiler, the LOAD MODULE statements can:

- Delete, display, or punch a load module
- Change the RMODE or the AMODE of a load module

Syntax

Syntax: MODIFY LOAD MODULE

```
►►─┬─ MODify ─┬─ LOAd MODule name is load-module-name ──────────────────────►
   └─ DELete ─┘

 ►─┬────────────────────────────────────────────┬───────────────────────────►
   └─ Version is ─┬─ version number ────────────┤
                  └─┬────────┬─┬─ HIGhest ─┬────┘
                    └─ NEXt ─┘ └─ LOWest ──┘
 ►─┬──────────────────────────┬──────────────────────────────────────────────►
   └─ user-specification ─────┘

 ►─┬──────────────────────────┬──────────────────────────────────────────────►
   └─ AMOde is ─┬─ 24 ──────┬─┘
                └─ ANY ◄────┘

 ►─┬──────────────────────────┬─────────────────────────────────────────────►◄
   └─ RMOde is ─┬─ 24 ──────┬─┘
                └─ ANY ◄────┘
```

Syntax: DISPLAY/PUNCH LOAD MODULE

```
►►─┬─ DISplay ─┬─ LOAd MODule name is load-module-name ──────────────────►
   └─ PUNch ───┘

 ►─┬─────────────────────────────┬───────────────────────────────────────────►
   └─ version-specification ─────┘

 ►─┬──────────────────────────────────────────────────────────────┬──────────►
   └─ PREpared by user-id ─┬──────────────────────────────────┬──┘
                           └─ PASsword is password ───────────┘

 ►─┬──────────────────────────────────────────────────────────┬──────────────►
   │ ┌──────────────────────────────────────────────────────┐ │
   │ │                       ┌─────────────────────────┐    │ │
   └─▼─┬─ WITh ──────────┬─▼─┬─ DETails ───┬───────────┴────┴─┘
       ├─ ALSo WITh ─────┤   ├─ HIStory ───┤
       └─ WITHOut ───────┘   ├─ ALL ───────┤
                             └─ NONe ──────┘

 ►─┬──────────────────┬─────────────────────────────────────────────────────►◄
   └─ WITh SYNtax ────┘
```

Parameters

LOAd MODule name is *load-module-name*

Identifies an existing load module. *Load-module-name* must be a 1- to 8-character alphanumeric value.

Version is

Supplies the version number of the load module. The version number defaults to the current session option for existing versions.

version-number

Specifies an explicit version number and must be an unsigned integer in the range 1 through 9999.

NEXt

Instructs the subschema compiler to assign the next highest or next lowest version number to *load-module-name*

HIGhest

Instructs the subschema compiler to assign the highest existing version number to *load-module-name*.

LOWest

Instructs the subschema compiler to assign the lowest existing version number to *load-module-name*.

user-specification

Identifies the user and the user's password. The default is the current session options.

If either the subschema compiler or the specific load module is secured, the compiler rejects the operation unless it finds the name and the password of an authorized user in one of the following places:

- The LOAD MODULE statement *user-specification* clause
- The current session option

Note: Expanded syntax for *user-specification* is presented in Chapter 13, "Parameter Expansions".

AMOde is ANY

Indicates that the module is invoked in 31-bit addressing mode. ANY is the default.

If RMODE is ANY, then AMODE must be ANY.

AMOde is 24

Indicates that the module is invoked in 24-bit addressing mode.

RMOde is ANY

Indicates that the module can be loaded above or below the 16-megabyte line. ANY is the default residency mode.

RMOde is 24

Indicates that the module must be loaded below the 16-megabyte line.

DETails

Display and punches load module length, entry point address, number of RLD (relocation directory) entries, security class, logical deletion flag, and module type (subschema).

HIStory

Display and punches the date and time the load module was created.

ALL

Display and punches the entire load module description.

NONe

Display and punches only the load module name and version.

WITh SYNtax

For PUNCH only, punches an object deck accompanied by the ADD LOAD MODULE syntax described in the *CA IDMS IDD DDDL Reference Guide*. This option is useful for producing an object deck that is to be placed in a load area other than the system load library.

Usage

Effect of DELETE on Load Modules

DELETE deletes the named load module from the load area of the dictionary. The subschema compiler also automatically erases the PROG-051 dictionary record occurrence associated with the load module, except if the record:

- Was not built by the subschema compiler
- Participates in other entity relationships, for example, maps

Effect of DISPLAY on Load Modules

DISPLAY displays online output at the terminal and lists batch output in the compiler's activity listing. The output always appears as comments regardless of the default option in effect.

Effect of PUNCH on Load Modules

PUNCH writes output to the system punch file or to a module in the dictionary. All punched output is also listed in a subschema compiler's activity listing.

The subschema compiler produces an object (relocatable) deck accompanied by ADD LOAD MODULE syntax from the named load module. The object deck can subsequently be link edited and placed in a load library. You can also use this option to move a load module from one dictionary to another.

Note: When you punch a load module from the dictionary load area (DDLDCLOD area) into an object module, the DDDL compiler omits the RMODE/AMODE attributes because the RMODE/AMODE clause is not acceptable to the linkage editor. If you are punching the load module to add it to a different dictionary, then you must edit the punched syntax to include the RMODE/AMODE clause.

Systems with 24-bit Addressing Load Modules Below the Line

For DC/UCF systems running in 24-bit mode, modules are loaded below the 16-megabyte lines regardless of the RMODE specification.

Residency Mode Determines Which Program Pool to Use

For DC/UCF systems running in 31-bit mode, modules with an RMODE of ANY are loaded into XA program pools (above the 16-megabyte line); modules with an RMODE of 24 are loaded into non-XA program pools (below the 16-megabyte line).

Examples

This example modifies the residency mode of load module DEHSS01:

```
modify load module name is dehss01
   rmode is any.
```

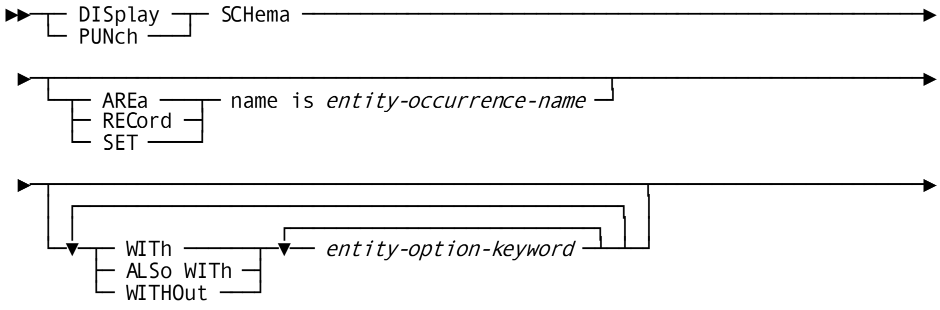
Note: For more information about defining load modules, see the LOAD MODULE statement in the *CA IDMS IDD DDDL Reference Guide*.

DISPLAY/PUNCH SCHEMA Statement

The DISPLAY and PUNCH SCHEMA statements produce as output the commented DDL statements that describe components of the schema that owns the current subschema.

Note: For a description of currency, see 9.7, “Establishing Schema and Subschema Currency”.

Syntax



```
►─┬─────────────────────────────────────────────────────────────────►
  └─ VERB ─┬─ ADD ─────┬┘
           ├─ MODify ──┤
           ├─ DELete ──┤
           ├─ DISplay ─┤
           └─ PUNch ───┘

►─┬─────────────────────────────────────────────────────────────────►◄
  └─ TO ─┬─ module-specification ─┬┘
         └─ SYSpch ───────────────┘
```

Parameters

SCHema

Displays or punches the commented description of the schema associated with the current subschema.

AREa name is *entity-occurrence-name*

Displays or punches the commented description of the named schema area entity.

FILe name is *entity-occurrence-name*

Displays or punches the commented description of the named schema file entity.

RECord name is *entity-occurrence-name*

Displays or punches the commented description of the named schema record entity.

SET name is *entity-occurrence-name*

Displays or punches the commented description of the named schema set entity.

entity-option-keyword

Names an option to be displayed or punched. The value of *entity-option-keyword* depends on the schema component. The following table lists values for *entity-option-keyword*.

Option	SCHEMA	AREA	RECORD	SET
ALL	x	x	x	x
ALL COMMENT TYPES	x		x	
AREAS	x		x	
ATTRIBUTES	x			
COMMENTS	x		x	
CULPRIT HEADERS	x		x	
DETAILS	x	x	x	x
ELEMENTS	x		x	

Option	SCHEMA	AREA	RECORD	SET
HISTORY	x			
NONE	x	x	x	x
OLQ HEADERS	x		x	
RECORDS	x			
SCHEMAS	x			
SETS	x			
SHARED STRUCTURES	x		x	
SUBSCHEMAS	x			
SYMBOLS	x	x	x	x
SYNONYMS	x		x	
USERS	x			

Example

This example displays the description of the DEPARTMENT record associated with the subschema's schema. Note that the subschema compiler produces commented output.

```
display schema record name is department without elements .
*+   ADD
*+   RECORD NAME IS DEPARTMENT
*+       SHARE STRUCTURE OF RECORD DEPARTMENT VERSION 100
*+       RECORD ID IS 410
*+       LOCATION MODE IS CALC USING ( DEPT-ID-0410 ) DUPLICATES ARE
*+             NOT ALLOWED
*+       WITHIN AREA ORG-DEMO-REGION OFFSET 2 PAGES FOR 48 PAGES
*+       RECORD NAME SYNONYM IS DEPARTMT FOR LANGUAGE ASSEMBLER
*+       .
```

Chapter 16: Writing Database Procedures

This section contains the following topics:

Database Procedures

Special-Purpose Subroutines

Database procedures are special-purpose subroutines designed to perform functions such as data compression and decompression. You write and compile these procedures as subroutines that are executed by the database management system whenever an access is made to an area or a record. User-written database procedures can be specified for non-SQL defined databases only.

Specifying a Procedure

Procedures are Called in the Schema

You specify as part of the schema definition when a procedure is to be called. At runtime, these procedures are called automatically; the call is transparent to the application program. You can specify that a procedure be called *before* or *after* any of the following DML statements or *on an error condition* resulting from execution of one of the commands in the following table.

Command	Description
READY	Prepares database areas for processing.
FINISH	Commits changes to the database and terminates the run unit.
COMMIT	Commits changes to the database.
ROLLBACK	Rolls back database changes and optionally terminates the run unit.
STORE	Adds a new record occurrence to the database.
CONNECT	Links a record occurrence to a set.

Command	Description
MODIFY	Changes the data content of an existing record occurrence.
DISCONNECT	Removes a member record occurrence from a set.
ERASE	Deletes a record occurrence from the database.
FIND	Locates a record occurrence in the database.
GET	Moves all data associated with a previously located record occurrence into the requesting program's variable storage.

The OBTAIN DML command combines the functions of the FIND and GET commands; thus, to perform a database procedure on an OBTAIN command, specify the procedure on a FIND and/or GET.

Common Uses of Database Procedures

Compression and Decompression

Data compression replaces repeating characters (most frequently blanks and binary zeros) and common character combinations with codes that decrease the amount of data stored in the database. Data decompression returns compressed data to its original form for use by an application program. A compression procedure (*IDMSCOMP*) and a decompression procedure (*IDMSDCOM*) are provided with CA IDMS/DB in source and object form.

The IDMSCOMP database procedure compresses record occurrences before storage, as follows:

- Converts repeating blanks into a 2-byte code
- Converts repeating binary zeros into a 2-byte code
- Converts other repeating characters into a 3-byte code
- Converts a number of commonly used character pairs into a 1-byte code

Data that does not fall into any of the above categories remains as is. Each group of as-is data is prefixed by a 2-byte length code.

CA IDMS/DB decompresses records after retrieval through the IDMSDCOM database procedure. These procedures are invoked automatically by CA IDMS/DB if you have coded the appropriate CALL parameters in the schema RECORD and AREA definitions.

Note: You can also use CA IDMS Presspack to compress data. For more information, see the *CA IDMS Presspack User Guide*.

Data Validation

Data validation involves checking data being stored to ensure that items :

- Are alphabetic or numeric
- Fall in user-specified ranges
- Are equal to specific values

If an item fails the check, the procedure can disallow storage of the record.

Privacy/Security

You can use database procedures to perform the following privacy/security functions:

- Encoding/decoding data to ensure physical data security
- Prohibiting programs from reading restricted data (record-occurrence level)
- Requiring passwords for access to restricted data (area level)
- Restricting use of a qualified ERASE DML command

Data Collection

Data collection procedures accumulate statistics and other information from areas and records being accessed.

Record Length for Variable-Length Native VSAM Records

Use the IDMSNVLR database procedure, provided with CA IDMS to transmit the length of a native VSAM variable-length record from an application program to the DBMS before a STORE or MODIFY DML command, and from the DBMS to the program after a GET DML command. The IDMSNVLR procedure is intended for use by programs accessing native VSAM variable-length records that do not contain an OCCURS-DEPENDING-ON field. IDMSNVLR allows the length to be communicated in the record's DBA-DEFINED-RDW (RECORD-DESCRIPTOR-WORD).

To use IDMSNVLR, the schema record description must provide for a standard DBA-DEFINED-RDW field (two-byte field plus two bytes of filler) as the last field in the record. The schema record description must be defined as follows:

```
record description.
    record name is record-name
    record id is record-id
    location mode is ...

    call idmsnvlr before store.
    call idmsvnlr before modify.
    call idmsvnlr after get.

    03  ...

    03  dba-defined-rdw
        comment 'this word is not maintained in the database.'
        05 rdw-len          pic S9(4) comp.
        05 filler           pic XX.
```

The DBA-DEFINED-RDW is not part of the physical record stored in the database. Before a STORE or MODIFY DML command is executed, IDMSNVLR strips off the DBA-DEFINED-RDW (the last four bytes of the record) by specifying RDW-LEN minus 4 as the record length. The DBA-DEFINED-RDW always includes the length of the DBA-DEFINED-RDW itself.

Before issuing a STORE or MODIFY DML command, the application program must move the length of the variable-length record (since that record was defined in the subschema) into the RDW-LEN. IDMSNVLR passes this value (minus 4) to the DBMS. After a GET DML command, IDMSNVLR returns the length of the subschema view of the record in the RDW-LEN field.

Although the schema description must specify the DBA-DEFINED-RDW as the last field of the record, the subschema description can specify the DBA-DEFINED-RDW as the first field of the record.

Coding Database Procedures

This section provides information to assist in writing database procedures.

You do not have to code or compile database procedures provided with CA IDMS/DB (for example, the IDMSNVLR procedure).

Considerations

There are two ways in which a database procedure can be invoked:

- It can be directly called by IDMSDBMS
- It can be called through a stub module that is called by IDMSDBMS.

Only fully reentrant assembler and LE-compliant COBOL and PL/I procedures can be invoked directly by IDMSDBMS. All other procedures must be called indirectly, at a cost in performance.

- Database procedures that are invoked directly by IDMSDBMS execute in system mode and MPMODE=ANY.
- For performance reasons, we recommend that all database procedures be written in fully reentrant assembler code.

Note: The methods that can be used for invoking a procedure depend on many factors including its language, calling conventions, reentrancy, and whether it issues CA IDMS DML Commands. For more information about the different methods for invoking procedures and how to choose one based on a procedure's characteristics, see 16.5, “Methods for Invoking Procedures”.

Issuing CA IDMS DML Commands in a Database Procedure

- While it is possible for database procedures to issue IDMS DML commands like navigational and SQL DML commands and commands that manipulate storage, scratch and queue resources, any such command can potentially result in a wait. A wait can result in deadlocks or degraded system performance because DBMS may be holding buffer locks when the procedure is called. If it is necessary to issue IDMS DML commands from within the procedure, consider the following:
  - Do not use the DBSTUB1 method described in B1 method described.
  - A directly-invoked assembler procedure must follow DC system mode calling conventions. For more information about DC system calling conventions and MPMODE, see "Calling Conventions for Numbered Exits" in the *CA IDMS System Operations Guide*.
  - A procedure that contains only IDMS DML commands associated with accessing a database can be compiled with a protocol of BATCH and execute in either a DC/UCF or a local mode address space. A COBOL or PL/I procedure that contains other (non-database access) IDMS DML commands must be compiled using the IDMS-DC protocol and can execute only in the DC/UCF address space. An assembler procedure can contain non-database access IDMS DML commands and execute in either environment provided the requested services are available. For example, requests for storage and scratch can be issued in either environment, whereas queue-related requests can only be issued within DC/UCF.
  - If the procedure accesses the database by binding a run unit or starting an SQL session, its database session is subordinate to that of the run unit under which the procedure is invoked. Therefore, actions such as FINISH or ROLLBACK that impact the invoking run-unit automatically have a similar impact on the procedure's database session if it is still active.
  - If the procedure accesses the same database as the invoking run-unit and database locks are being maintained, deadlocks between the two sessions are possible unless they are made to share the same transaction, see "Sharing Transactions Among Sessions" in the *CA IDMS Navigational DML Programming Guide*.
- Avoid using operating system functions that may cause the central version region to wait. This degrades performance.
- Ensure the module name is the name specified in the schema CALL statement. Database procedures are no longer linked with subschema modules. They are dynamically loaded by DBMS on the first call. The entry point name can be different from the module name.

Area Procedures

You must write *area procedures* to accept the following five blocks of information which are passed when the database procedure is executed by CA IDMS/DB:

- Procedure control block (20 bytes)
- Application control block (236 bytes)
- Application program information block (user-specified length)
- Area control block (28 bytes)
- IDMS statistics block (100 bytes)

Record Procedures

You must write *record procedures* to accept the following five blocks of information which are passed when the database procedure is executed by CA IDMS/DB:

- Procedure control block (20 bytes)
- Application control block (236 bytes)
- Application program information block (user-specified length)
- Record control block (56 bytes)
- Record occurrence block (length specified in schema)

Record procedures have access to the entire data portion of the schema-defined records. They are not restricted to the subschema views seen by application programs.

Database Procedure Blocks

The following tables show the format of the database procedure blocks.

Procedure Control Block

This is the first block of information passed to both area and record procedures. It contains information that reflects the general conditions under which the database procedure is being invoked. Total length is 20 bytes.

Item	Usage	Length	Description
Entry Level	Alphanumeric	4 bytes	Level at which the procedure is invoked: REC or AREA
Entry Time	Alphanumeric	4 bytes	The time the procedure is invoked: BFOR, AFTR, or ERR

Item	Usage	Length	Description
Major Code	Alphanumeric	2 bytes	Major DML code of the DML command for which the procedure is being invoked (that is, 12 for STORE, or 03 for FIND, and so forth)
IDBMSCOM Code	Binary	2 bytes	IDBMSCOM code of the DML command for which the procedure is being invoked (that is, 14 for FIND NEXT WITHIN SET, or 15 for FIND NEXT WITHIN AREA, and so forth)
Cancel Indicator	Binary	2 bytes	Zero indicates that the DML command should be performed; nonzero requests cancellation of the DML command. The initial value of zero can be reset by a BEFORE procedure.
Record Indicator	Binary	1 byte	Indicates whether record is present in the Record Occurrence Block 0 - record is not present 1 - record is present
Filler	Alphanumeric	1 byte	Reserved
User Item	Binary	4 bytes	For user storage, as needed (normally, an address); initialized to zero. This value is preserved across calls to the procedure.

Application Control Block

This is the second block of information passed to both area and record procedures. It contains information that reflects the status of the application program at procedure execution time. Total length is 236 bytes.

Item	Usage	Length	Description
Subschema Name	Alphanumeric	8 bytes	Name of subschema being used
Program Name	Alphanumeric	8 bytes	Name of application program
Error-Status Indicator	Alphanumeric	4 bytes	Major DML code (first two bytes) of the command for which the procedure is being invoked, and the minor error-status code (second two bytes)

Item	Usage	Length	Description
Database Key	Binary	4 bytes	The database key that is current of run unit
Record Name	Alphanumeric	18 bytes	Name of record type that is current of run unit
Area Name	Alphanumeric	18 bytes	Name of area to which current of run unit is assigned
Filler	Alphanumeric	18 bytes	Reserved for future use
Error-Set Name	Alphanumeric	18 bytes	Name of error-set type, if applicable
Error-Record Name	Alphanumeric	18 bytes	Name of error-record type, if applicable
Error-Area Name	Alphanumeric	18 bytes	Name of error area, if applicable
IDBMSCOM Array	Alphanumeric	100 bytes	System IDBMSCOM array for passing function information
Direct Db-key	Binary	4 bytes	Item used by application program to specify a database key for storing a record in DIRECT storage mode

Application Program Information Block

This is the third block of information passed to both area and record procedures. It contains information (if any) passed between the application program and database procedure. Total length is determined by user.

Item	Usage	Length	Description
Application Program Information	DBA-defined	DBA-defined	Information passed from application program using a BIND PROCEDURE statement; if not used, this field must be defined as a 4-byte alphanumeric item

Area Control Block

This is the fourth block of information passed to area procedures. It contains information about the area for which the procedure is being invoked. Total length is 28 bytes.

Item	Usage	Length	Description
Area Name	Alphanumeric	18 bytes	Name of area for which DML command is being invoked
Filler	Alphanumeric	2 bytes	
Low Page	Binary	4 bytes	Number of lowest page in area
High Page	Binary	4 bytes	Number of highest page in area

CA IDMS Statistics Block

This is the fifth block of information passed to area procedures. It contains runtime statistics for the application program (same statistics obtained by the DML command ACCEPT IDMS-STATISTICS). Total length is 100 bytes.

Item	Usage	Length	Description
Date	Alphanumeric	8 bytes	Today's date in the format *mm/dd/yy*
Time	Alphanumeric	8 bytes	The time of the last occurrence of BIND RUN-UNIT, FINISH, or run-unit abort; in the format *hhmmsshh*
Pages Read	Binary	4 bytes	Total pages read by application program
Pages Written	Binary	4 bytes	Total pages written by application program
Pages Requested	Binary	4 bytes	Total pages requested by application program
CALC Records	Binary	4 bytes	Total CALC records stored with no overflow
CALC Overflow	Binary	4 bytes	Total CALC records that overflowed
VIA Records	Binary	4 bytes	Total VIA records stored with no overflow
VIA Overflow	Binary	4 bytes	Total VIA records that overflowed from target page

Item	Usage	Length	Description
Records Requested	Binary	4 bytes	Total number of records accessed by the DBMS
Records Current	Binary	4 bytes	Total number of records established as current of run unit
Calls to CA IDMS/DB	Binary	4 bytes	Total calls made for DBMS services
Fragments Stored	Binary	4 bytes	Total variable length record fragments
Records Relocated	Binary	4 bytes	Total records relocated
Locks Requested1	Binary	4 bytes	Total number of record locks requested
Select Locks Held2	Binary	4 bytes	Number of shared locks now held
Update Locks Held2	Binary	4 bytes	Number of exclusive locks now held
Run Unit Id2	Binary	4 bytes	LID: Local Identification number of transaction for journaling purposes; incremented by one and carried across central versions until the journal is reinitialized
Task Id2	Binary	4 bytes	Identification number of central version task; reinitialized for each central version run and incremented by 1, beginning at 2 (0 and 1 are reserved for system)
Local Identification 2	Alphanumeric	8 bytes	Identification code of batch or TP program to facilitate location of dumps and elements in the central version log
Filler	Alphanumeric	8 bytes	Reserved

1. As a lock is released, this value is not decremented

2. Applies to central version only

Record Control Block

This is the fourth block of information passed to record procedures. It contains information regarding the record type for which the procedure is being invoked. Total length is 56 bytes.

Item	Usage	Length	Description
Record Name	Alphanumeri c	18 bytes	Name of record type for which DML command is being invoked
Area Name	Alphanumeri c	18 bytes	Name of area to which record is assigned
Record ID	Binary	2 bytes	Identification number of record type for which DML command is being invoked
Record Length	Binary	2 bytes	Length (data only), in bytes, of record
Control Length	Binary	2 bytes	Length (data only), in bytes, of record up to and including the last CALC or sort-control field
Maximum Length	Binary	2 bytes	Actual length of fixed-length record or maximum length of variable-length record, in bytes
Database Key	Binary	4 bytes	Database key of record
Low Page	Binary	4 bytes	Number of lowest page on which records of this type can exist
High Page	Binary	4 bytes	Number of highest page on which records of this type can exist

Record Occurrence Block

This is the fifth block of information passed to record procedures. It is used to pass the actual record occurrence for which the procedure is invoked. There are situations in which the record occurrence is not available to be passed to the procedure. Total length is defined in the record type's schema description.

Item	Usage	Length	Description
Record Occurrence	As defined in schema	As defined in schema	Actual record that is the target of the DML command

Whenever possible, the record occurrence for which the procedure is being invoked is passed, but under some conditions the DBMS may not have immediate access to this data. In all cases, the fifth parameter is passed to the procedure, but its validity is not guaranteed under all scenarios. The following table indicates the availability of the record occurrence block data relative to procedure call times:

DML Verb	Procedure Call Times BEFORE	Procedure Call Times AFTER	Procedure Call Times ON ERROR
CONNECT	Present if last good verb was an OBTAIN, STORE, or MODIFY for target record type, else uncertain	Present if last good verb was an OBTAIN, STORE, or MODIFY for target record type, else uncertain	Present if last good verb was an OBTAIN, STORE, or MODIFY for target record type, else uncertain
DISCONNECT	Present if last good verb was an OBTAIN, STORE, or MODIFY for target record type, else uncertain	Present if last good verb was an OBTAIN, STORE, or MODIFY for target record type, else uncertain	Present if last good verb was an OBTAIN, STORE, or MODIFY for target record type, else uncertain
ERASE	Present if last good verb was an OBTAIN, STORE, or MODIFY for target record type, else uncertain	Present if last good verb was an OBTAIN, STORE, or MODIFY for target record type, else uncertain	Present if last good verb was an OBTAIN, STORE, or MODIFY for target record type, else uncertain
FIND	Unavailable unless access is CALC, then calckey fields are available	Available if FIND executed as part of OBTAIN else unavailable. If access is CALC, calckey fields are available	Unavailable unless access is CALC, then calckey fields are available
GET	Available if GET executed as part of an OBTAIN, else unavailable	Available	Available if GET executed as part of an OBTAIN, else uncertain
MODIFY	Available, contains data passed from user program	Available, contains data passed from user program	Available, contains data passed from user program
STORE	Available, contains data passed from user program	Available, contains data passed from user program	Available, contains data passed from user program

Establishing Communication Between Programs and Procedures

Program/Procedure Communication

Some database procedures may require specific information from the calling application program (for example, a password for a security routine). Use the application program information block to pass this information. Using the BIND PROCEDURE DML command, the programmer binds space in program variable storage for the information to be passed. At program runtime, whenever the procedure is called, the information in the program space bound to the procedure is placed in the procedure's application program information block.

Executing Under the Central Version in a Different Address Space

If the application program is executing under the central version and in a different address space, the program must bind a 256-byte space in variable storage. Programs running in the same address space as the central version or in local mode can bind a variable amount of space, but 256 bytes is recommended in case of future changes in the operating configuration.

In the central version environment, the BIND procedure DML has the function of passing the information in the application program information block to the database procedure. To get information back from the database procedure, the application program should issue an ACCEPT...FROM...PROCEDURE DML statement. If the application program wishes to send new information to the database procedure, the application program should alter the data in the application program information block and then issue another BIND procedure DML statement, which will cause the central version's copy of the application program information block to be refreshed.

No Information Passed

Usually, no information is passed between the program and the database procedure, since database procedures are normally transparent to application programs. When no information is passed, the database procedure must define the application program information block as a 4-byte alphanumeric item.

Specifying When to Call Database Procedures

Using CALL

To specify when a database procedure is to be called at runtime, you use the CALL statement in the schema DDL for areas and records. You can use database procedures with any number of DML commands for any number of areas or records. For example, to compress/decompress JOB records with the CA IDMS/DB-supplied procedures, specify the following CALL statements for the JOB record type:

```
call idmscomp before store.

call idmscomp before modify.

call idmsdcom after get.
```

Note: If the schema contains any records for which IDMSCOMP or IDMSDCOM is called, IDMSDCOMP and IDMSDCOM must be called as area procedures 'BEFORE FINISH' and 'BEFORE ROLLBACK' to release the storage used for internal compression/decompression work areas.

Link Editing Database Procedures

You must link database procedures as standalone modules. Database procedures linked with subschema modules are no longer supported.

Procedures Written in COBOL under z/VSE

For database procedures written in COBOL that will execute under z/VSE, assemble the following CSECT and catalog it into the appropriate relocatable library:

```
ILBDMNS0  CSECT
     DC X'FF'
     END
```

Assign to the CSECT a library member name other than ILBDMNS0 so that the CSECT will not be linked to all COBOL programs. This CSECT name must be included in the link edit of the COBOL database procedure. The procedure checks the field contained in this CSECT to establish the appropriate linkage with CA IDMS/DB.

Executing Database Procedures

When a DML command is issued at application run time, all BEFORE procedures are executed in the order specified in the schema. A BEFORE procedure can prevent execution of the DML command in either of the following ways:

- By resetting the cancel indicator in the procedure control block to a nonzero value
- By resetting the error-status indicator in the application control block to a nonzero value

The DML command is not executed if either of the above conditions exists when all BEFORE procedures have been completed. If the cancel indicator in the procedure control block is reset to a nonzero value, control passes directly to the AFTER procedures. If the error-status indicator in the application control block is reset to a nonzero value, control passes directly to the ON-ERROR procedures.

Note: To prevent execution of a FINISH DML command, a BEFORE FINISH procedure must reset the error-status indicator to a nonzero value. You cannot use the cancel indicator for this purpose.

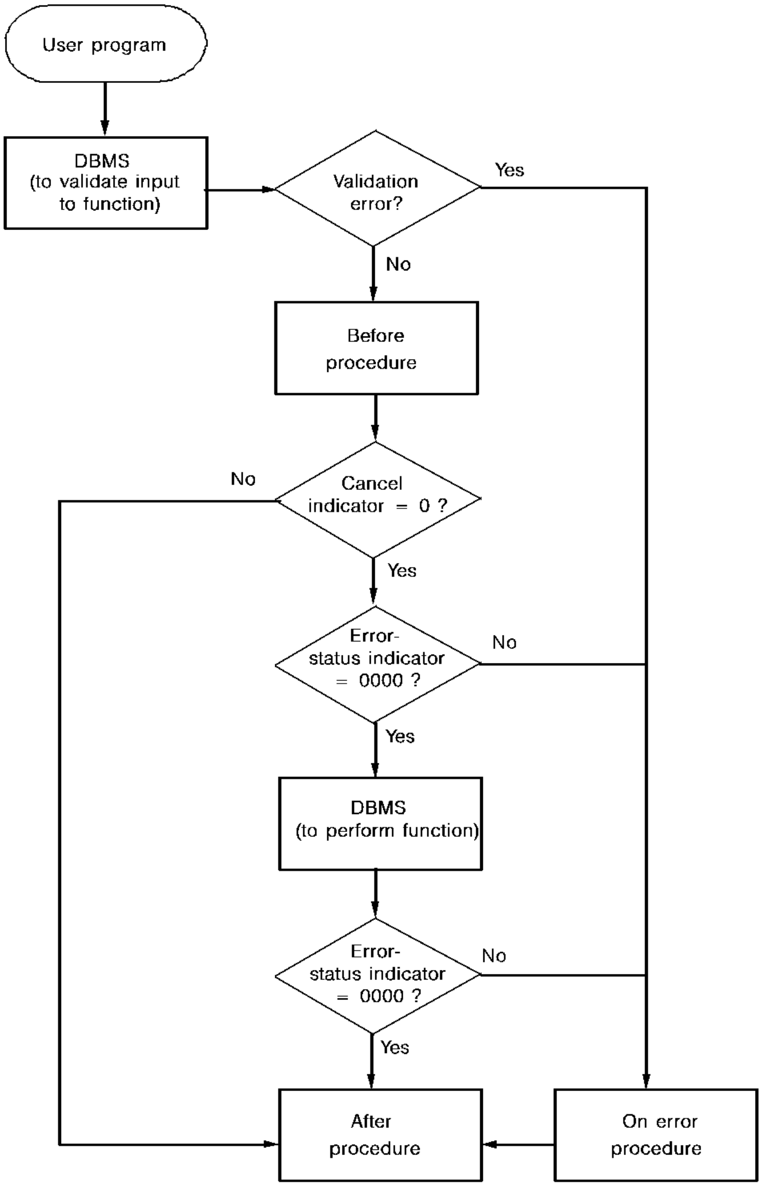
Resetting the Error-Status Indicator

In resetting the error-status indicator, the procedure should change only the last two bytes (the minor code); the procedure should leave the first two bytes (the major code) unchanged. However, when the value of the error-status indicator is zero, the procedure should reset the indicator with the value from the major code item of the procedure control block.

Note: To avoid confusion, user-defined error-status codes should not duplicate CA IDMS/DB error-status codes.

At this point, if the DML command has not been canceled, the command is executed. *ON-ERROR procedures* are executed if errors have occurred during validation by the DBMS or if the error-status indicator contains a value other than 0000. If, because of validation errors, execution immediately drops through to an ON-ERROR procedure, BEFORE procedures and the DML command itself are not performed. The error-status indicator can be reset by either a BEFORE procedure or the DML command.

If the error-status is 00, any AFTER procedures are now executed. However, if the error-status is not 00, AFTER procedures are not executed unless at least one ON-ERROR procedure has been defined for the verb. Because AFTER procedures can be executed when the DML command has been suppressed or a non-zero error-status has been returned they should always check the values of the cancel indicator and error-status indicator.



Methods for Invoking Procedures

A database procedure is called as an extension of the database engine. A procedure can be called directly by IDMSDBMS. This is referred to as the Direct invocation method. Alternatively, a procedure can be invoked indirectly by using one of two techniques referred to as DBSTUB1 and DBSTUB2. These are described later in this section.

The methods that can be used to invoke a given procedure depend on several factors:

- Language of the procedure
- Reentrancy or LE-compliance
- Calling conventions that it uses
- Whether it issues IDMS DML requests

The following table identifies the methods that can be used for invoking procedures with differing characteristics. Where multiple invocation methods are listed as valid, the recommended method is highlighted.

Language	Comments	IDMS DML Issued by Procedure	Valid Invocation Methods
Assembler	Reentrant, DC calling conventions	Does not matter	**Direct**, DBSTUB1, DBSTUB2
Assembler	Non-reentrant, DC calling conventions	No	**DBSTUB1**
Assembler	Reentrant, IBM calling conventions	No	**DBSTUB1**, DBSTUB2
Assembler	Non-reentrant, IBM calling conventions	No	**DBSTUB1**, DBSTUB2
Assembler	Reentrant, IBM calling conventions	Yes	**DBSTUB2**
VS Cobol	Non-LE-compliant	No	**DBSTUB1**, DBSTUB2
VS Cobol	Non-LE-compliant, reentrant	Yes	**DBSTUB2**
VS Cobol/2	Non-LE-compliant, reentrant	Does not matter	**DBSTUB2**
LE-compliant Cobol	LE-compliant, reentrant	Does not matter	**Direct**, DBSTUB2
PL/I	Non-LE-compliant, reentrant	Does not matter	**DBSTUB2**

Language	Comments	IDMS DML Issued by Procedure	Valid Invocation Methods
PL/I	LE-compliant, reentrant	Does not matter	**Direct**, DBSTUB2

DBSTUB1 Invocation Method

The DBSTUB1 invocation method is valid only for calling a COBOL program compiled with VS-COBOL or a non-reentrant assembler program. It is *not* valid for any other program such as a program written in COBOL, PL/I or any LE-compliant language. DBSTUB1 is an assembler front end that is linked with the actual procedure. The linked module name must match the name in the schema CALL statement. The entry point must point to DBSTUB1's entry point.

DBSTUB1 is written with DC system calling conventions. It runs in MPMODE=DB, which means that it holds a lock on MPMODE DB when it gains control. No other program that runs in MPMODE DB can run at the same time. Once it has control, it calls the database procedure that is linked with it. The procedure called must not issue any IDMS calls because during such a call, the MPMODE DB lock protection is lost.

Note: For more information about DC system calling conventions and MPMODE, see "Calling Conventions for Numbered Exits" in the *CA IDMS System Operations Guide*.

A unique DBSTUB1 must be written for and linked with each database procedure that needs this interface. Usually only the entry point that is called must be changed.

Following is a sample of DBSTUB1 that calls the CHECKIT database procedure.

```
DBSTUB1  TITLE 'Example of a DB procedure'
         #MOPT CSECT=DBSTUB1,ENV=SYS
*
*    The following code shows how a COBOL database procedure might
*    be called in a multi-tasking environment.  This program is
*    linked with the COBOL procedure.  The module name must be the
*    same as the name coded in the Schema CALL statement.
*    The entry point is STUBEP1.
*
*    The following code emulates how DBMS calls DB procedures.
*    When this procedure receives control the task is single
*    threaded on the MPMODE=DB lock.
*
*    On Entry: R1 already points to plist.
*
```

```
        USING CSA,R10
STUBEP1  #START MPMODE=DB
         L      R15,=V(CHECKIT)        Base linked DB Procedure.
         CLC    =X'4700',0(R15)        Bif DC mode prog.
         BE     STUB010
*
         #CHKSTK =(18+1)               Make sure room on stack,
*                                           for the savearea.
         BALR   R14,R15                Call Standard mode program.
         B      STUB020
*
STUB010  #CALL  (R15)                  Call DC mode program.
*
STUB020  #RTN                          Return to DBMS.
         LTORG
         COPY   #CSADS
         END
```

Following is how DBSTUB1 would be linked with CHECKIT.

```
INCLUDE OBJLIB(DBSTUB1)
INCLUDE OBJLIB(CHECKIT)
ENTRY   STUBEP1
MODE    AMODE(31),RMODE(ANY)
NAME    CHECKIT(R)
```

DBSTUB2 Invocation Method

The DBSTUB2 invocation method is valid for calling a program written in COBOL or PL/I compiled with any compiler supported by CA IDMS. Usage of this method is optional and *not* recommended when the program is compiled with an LE-compliant compiler because of performance. This program is an assembler front end that is linked separately from the procedure it calls. The DBSTUB2 program must be linked as the name specified in the Schema CALL statement. The database procedure must be linked as a second name and defined to DC in the SYSGEN.

DBSTUB2 is written with DC system mode calling convention. It runs in MPMODE=CALLER which means multiple task threads can be running through it at the same time and this code must be totally reentrant.

Once DBSTUB2 gains control it activates the real procedure with a #LINK command. IDMS/DC will setup and call the program in user mode.

When the procedure ends, control is returned to DBSTUB2.

This method violates the principal of maintaining control of the CPU. One or more #WAITs can occur during the execution of the #LINK. This increases the likelihood of deadlocks or performance problems.

Following is a sample of DBSTUB2 that calls the CHECKIT database procedure. However, in this case DBSTUB2 has been linked as CHECKIT and the COBOL CHECKIT has been linked as CHECKIT2.

```
DBSTUB2  TITLE 'Example DB procedure'
         #MOPT CSECT=DBSTUB2,ENV=SYS
*
*    The following code shows how a database procedure might call
*    a program written in a high level language like COBOL II.
*
*    The name in the Schema CALL statement must be this module.
*    The module this program #LINKs must be defined in the DC
*    SYSGEN.  In this example the CHECKIT database procedure
*    would have been renamed to CHECKIT2 and this procedure
*    would be called CHECKIT.
*
*    By #LINKing the DB procedure, the current system mode
*    environment is preserved.  The #LINKed subprogram is setup
*    and called based on how it is defined to DC.  For example
*    a COBOL program would get called as a quasi-reentrant with
*    DC allocating the private copy of WORKING-STORAGE for it.
*
*    ON Entry:  R1 points to the procedure parmlist
*
         USING CSA,R10
STUBEP1  #START MPMODE=CALLER
*        #GETSTK  =8,REG=R11     get 8 words for plist
         USING LWA,R11
*
         LM    R3,R7,0(R1)        get db parameters
         #LINK PGM='CHECKIT2',PARMS=((R3),(R4),(R5),(R6),(R7)),             X
               PLIST=SYSPLIST     link to DB procedure
 *
          #RTN                     return to the DBMS
          LTORG
 LWA      DSECT                    local work area
 SYSPLIST DS    8F                 PLIST for #LINK
          COPY  #CSADS
          END
```

Following is how DBSTUB2 would be linked.

```
INCLUDE OBJLIB(DBSTUB2)
ENTRY   STUBEP1
MODE    AMODE(31),RMODE(ANY)
NAME    CHECKIT(R)
```

CHECKIT would get linked:

```
INCLUDE OBJLIB(CHECKIT)
ENTRY   CHECKIT
MODE    AMODE(31),RMODE(ANY)
NAME    CHECKIT2(R)
```

In Conclusion

The methods described in this section solve the problem of calling database procedures that are not reentrant or are written in COBOL or PL/I and compiled with a non-LE-compliant compiler. The overhead of using the methods will be high. DBSTUB1 will have less overhead than DBSTUB2. For performance reasons, we recommend avoiding these methods by writing database procedures as fully reentrant assembler programs. The second best option is to write a database procedure in COBOL or PL/I and compile it with an LE-compliant compiler.

Considerations for Non-Reentrant or Non-LE-Compliant Database Procedures

Invoking Non-Reentrant or Non-LE-Compliant Procedures

There are special considerations for invoking these types of procedures.

- In a multi-tasking environment, a non-reentrant database procedure written in assembler can only be called indirectly using the DBSTUB1 approach described in this section.
- Database procedures written in COBOL or PL/I that are compiled with a non-LE-compliant compiler (such as, COBOL/II, PL/I 2.3) must be called indirectly using the DBSTUB2 approach described in this section.

  Note: LE is the abbreviation for Language Environment.

Database Procedure Example

Using the Employee database, a company uses a database procedure to perform validity checks on employee identification numbers (ID-0415) before EMPLOYEE records are stored in the employee database. A COBOL program CHECKID functions as follows:

- Describes (in the program's LINKAGE SECTION) the five blocks of information that CA IDMS/DB passes to all database procedures
- Performs the validity checks by using the first four bytes of the EMPLOYEE record, as passed to the program's record occurrence block
- Sets the error-status indicator in the application control block to 99 if the employee id (ID-0415) fails validity checks

Sample Database Procedure

The LINKAGE SECTION describes the five blocks of information that CA IDMS/DB passes to the procedure. ID-0415 (employee ID) is the first four bytes of the record occurrence passed to the procedure. If ID-0415 does not pass the validity check, the error-status indicator in the application control block is set to 99 to prevent execution of the DML command for which the procedure was called.

Sample database procedure

```
**************************************************************
 IDENTIFICATION DIVISION.
**************************************************************

 PROGRAM-ID.           CHECKID.
 DATE-WRITTEN.         JUNE 15, 1991.
 AUTHOR.               COMMONWEATHER CORP.
 REMARKS.              VALIDATES INCOMING EMPLOYEE NUMBERS.

**************************************************************
 ENVIRONMENT DIVISION.
**************************************************************

**************************************************************
 DATA DIVISION.
**************************************************************
```

```
LINKAGE SECTION.

 01  PROC-CTRL.
     02  PC-ENTRY-LEVEL           PIC X(4).
     02  PC-ENTRY-TIME            PIC X(4).
     02  PC-MAJOR-CODE            PIC XX.
     02  PC-IDBMSCOM-CODE         PIC 9(4) COMP.
     02  PC-CANCEL-SWITCH         PIC 9(4) COMP.
     02  FILLER                   PIC XX.
     02  PC-USER-AREA             PIC 9(8) COMP.

 01  APPLIC-CTRL.
     02  SC-SUB-NAME              PIC X(8).
     02  SC-PROG-NAME             PIC X(8).
     02  SC-ERROR-STATUS.
         03  SC-ERR-MAJOR         PIC XX.
         03  SC-ERR-MINOR         PIC XX.
     02  SC-DBKEY                 PIC 9(8) COMP.
     02  SC-REC-NAME              PIC X(18).
     02  SC-AREA-NAME             PIC X(18).
    02 FILLER                    PIC X(18).
     02  SC-ERR-SET-NAME          PIC X(18).
     02  SC-ERR-REC-NAME          PIC X(18).
     02  SC-ERR-AREA-NAME         PIC X(18).
     02  SC-IDBMSCOM              PIC X(100).
     02  SC-DIRECT-DBKEY          PIC 9(8) COMP.
 01  A-P-COMM-DATA               PIC X(4).
 01  REC-CTRL-BLOCK.
     02  RC-REC-NAME             PIC X(18).
     02  RC-AREA-NAME            PIC X(18).
     02  RC-REC-ID               PIC 9(4) COMP.
     02  RC-REC-LENGTH           PIC 9(4) COMP.
     02  RC-REC-CTRL-LEN         PIC 9(4) COMP.
     02  RC-REC-MAX-LEN          PIC 9(4) COMP.
     02  RC-DBKEY                PIC 9(8) COMP.
     02  RC-LPL                  PIC 9(8) COMP.
     02  RC-HPL                  PIC 9(8) COMP.

 01  EMPLOYEE.
     02  ID-0415                 PIC X(4).
     02  FILLER                  PIC X(103).

*************************************************************
 PROCEDURE DIVISION USING        PROC-CTRL
                                 APPLIC-CTRL
                                 A-P-COMM-DATA
                                 REC-CTRL-BLOCK
                                 EMPLOYEE.
*************************************************************

     IF ID-0415 NOT NUMERIC
     OR ID-0415 LESS THAN '0001'
     OR ID-0415 GREATER THAN '9999'
     THEN MOVE 99 TO SC-ERR-MINOR.
     GOBACK.
```

Schema Statement

Include the following clauses in the record description for EMPLOYEE in the Employee schema:

```
CALL CHECKID BEFORE STORE.
CALL CHECKID BEFORE MODIFY.
```

Any program using a subschema compiled under this schema automatically invokes the database procedure CHECKID before storing or modifying an EMPLOYEE record occurrence.

Chapter 17: Allocating and Formatting Files

This section contains the following topics:

Making Files Accessible to CA IDMS/DB

Steps

To make a file accessible to CA IDMS/DB, follow these steps:

1. Use physical DDL statements to: define the file within a new or existing segment and associate it with one or more new or existing areas; include the segment definition, with any file and/or area overrides, in a DMCL.
2. Make available the DMCL in which the file's segment is included.
3. Create the file using facilities provided by your operating system.
4. Format the file.

This chapter describes steps 3 and 4.

Types of Files

Available Options

CA IDMS/DB can access data stored in the following types of files:

File Type	Access Method	File Structure
Direct access	EXCP (z/OS, z/VSE)	A file block corresponds to a database page
Physical sequential	EXCP (z/OS) SAM (z/VSE)	A file block corresponds to a database page

File Type	Access Method	File Structure
CMS format minidisk	DASD block I/O (z/VM)	A file block corresponds to a database page
VSAM database	VSAM (z/OS, z/VSE)	An ESDS VSAM file in which each Control Interval contains a single database page plus 8 bytes of control information used by VSAM
Native VSAM	VSAM (z/OS, z/VSE)	An ESDS, KSDS, or RRDS VSAM file or PATH in which each VSAM record corresponds to an IDMS record

Specifying the File Type in the FILE Statement

When you define a file using a physical DDL FILE statement, you specify the file's type using these parameters:

FILE Statement Parameter	Corresponding File Type
NONVSAM or BDAM	Direct access (z/OS, z/VSE) Physical sequential (z/OS) CMS format minidisk (z/VM)
VSAM	VSAM database (z/OS, z/VSE)
ESDS KSDS RRDS PATH	Native VSAM (z/OS, z/VSE)

File Access Methods

Determines How CA IDMS/DB Gains Access to Files

When an application program issues a call to CA IDMS/DB for retrieval or storage of a record or row of data, CA IDMS/DB maps the database page that contains the record or row to the corresponding block or blocks in the file. The means by which this mapping occurs varies according to the access method in use:

- EXCP (z/OS, z/VSE)
- SAM (z/VSE)
- DASD Block I/O (z/VM)
- VSAM (z/OS, z/VSE)

EXCP Access Method

The EXCP access method is used in z/OS and z/VSE to take advantage of extended addressing. Using EXCP as an access method, CA IDMS/DB maps the database page number to a relative track and record number. The database page size must equal the block size of the file.

SAM Access Method

Using SAM as an access method, CA IDMS/DB maps the first database page number to a relative block number (RBN) within the sequential access file. It then reads forward sequentially from that RBN. The database page size must equal the block size of the file.

DASD Block I/O

In z/VM, all CA IDMS/DB files are allocated as separate minidisks and are accessed using DASD Block I/O.

Note: For more information, see the *CA IDMS Installation and Maintenance Guide—z/VM*.

VSAM Access Method

CA IDMS/DB can take advantage of extended addressing when accessing data by means of the VSAM access method. All VSAM macros use the AMODE=31 and RMODE=31 parameters. Therefore, all VSAM control blocks are allocated above the 16-megabyte line.

Accessing VSAM Database Files

Using VSAM as an access method to VSAM database files, CA IDMS/DB maps the database page number to a VSAM control interval and issues a request to VSAM for that control interval.

Accessing Native VSAM Files

Existing VSAM files to be accessed by CA IDMS/DB are referred to as *native VSAM files* because they are not formatted into pages as is the case with all other file types. CA IDMS/DB accesses native VSAM files using VSAM record-level services. A native VSAM file can have one of the following structures:

- Key-sequenced (KSDS)
- Entry-sequenced (ESDS)
- Relative record (RRDS)

Regardless of the type of file being accessed, each is represented by a single record type described to CA IDMS/DB in a non-SQL schema definition.

Note: For more information, see 17.6, “Considerations for Native VSAM Files".

Choosing Between VSAM and Non-VSAM File Types

In z/OS and z/VSE, you may define database files as either VSAM or non-VSAM.

z/VSE: To define non-VSAM files on FBA disk devices (type 3310 or type 3370), use a sequential label (that is, an SD attribute on the DLBL statement).

Creating Disk Files

Use Operating System Facilities

Use facilities provided by your operating system to create and catalog the files.

File Placement on Disk

You can reduce I/O response time by planning where you place files on a disk. In general, spread high activity files across disk devices and channels. Particularly, consider the placement of disk journal files used by systems engaged in high-volume update activity.

Multi-volume Files

CA IDMS does not support files that span multiple physical volumes. If an area is too large to reside on a single volume, it must be mapped to multiple files, each residing on a single physical volume.

Parallel Access Volume Exploitation

Parallel Access Volumes (PAV devices) allow concurrent I/O against individual files. If a database is allocated on a PAV device, it may reduce I/O wait times and thus increase transaction throughput and improve response times. Similar, although less significant, benefits may be achieved by allocating a journal file on a PAV device since I/O contention with the journal archive utility may be reduced.

No special action is needed to exploit this feature beyond allocating a database or journal file on a properly configured PAV device.

Valid Disk Devices for Archive and Tape Journal Files

The following table summarizes the disk device types CA IDMS/DB supports for archive and tape journal files:

System	Device Types
z/OS	Any supported by QSAM
z/VSE	Any supported by SAM
z/VM	Any supported by QSAM

Valid Device Types for Disk Journal Files and Database Files

The following table summarizes the device types CA IDMS/DB supports for disk journal files and database files:

System	Device Types
z/OS	Any supported by BDAM or VSAM
z/VSE	Any supported by SAM
z/VM	Any supported by DASD Block I/O

Maximum Area Page Sizes

When allocating non-VSAM files in z/OS and z/VSE operating systems, the page size of an area is restricted by the track size of the disk device being used. The following table identifies the maximum page size for non-VSAM files in z/OS and z/VSE operating systems:

Disk Device	Maximum Page Size	Bytes per Track
2311	3624	3625
2314	7292	7294
2321	2000	2092
3330/3330B	13028	13030
3340	8368	8535
3350	19068	19254
3375	32764	36000
3380	32764	47476
3390	32764	56664

File Characteristics

Non-VSAM Files in z/OS

To create a non-VSAM file in z/OS, use a JCL statement or a facility such as TSO. The DCB characteristics of the file must be:

Parameter	Value
DSORG	PS or DA
BLKSIZE	Page size of the area(s) mapped to the file
RECFM	F

VSAM Files

To create a VSAM database or journal file, you use the IDCAMS utility from IBM. The following IDCAMS statements are used:

- DEFINE SPACE—Allocates disk space for one or more VSAM files; alternatively, the database file can be defined in its own data space
- DEFINE CLUSTER—Creates the database file as an ESDS VSAM cluster specifying the following attributes:

Attribute	Description
RECORDS	Assign: ■ PRIMARY SPACE as the number of pages mapped to the file SECONDARY SPACE as the value 2
RECORDSIZE	Assign: ■ AVERAGE as the page size of the area mapped to the file MAXIMUM as the page size of the area mapped to the file
CONTROL INTERVALSIZE	■ For database files,assign a value at least 8 bytes larger than the page size of the area mapped to the file, but less than twice the page size minus 8 ((2 * page size)- 8) ■ For disk journal files, assign a value that is the same as the page size of the journal buffer
SHAREOPTIONS	Assign (3 3)
REUSE SUBALLOCATE or UNIQUE	
NONSPANNED	
NONINDEXED	

More Information

- For more information about creating CMS-format minidisks to be used by CA IDMS/DB, see the *CA IDMS Installation and Maintenance Guide—z/VM*.
- For more information about defining and accessing native VSAM files, see 17.6, “Considerations for Native VSAM Files".

Formatting Files

What Formatting Means

Formatting means initializing database or disk journal files into database pages or blocks according to information provided by the DMCL.

Important! NEVER format native VSAM files

Formatting Database Files

When you issue a FORMAT command against a database file, CA IDMS/DB:

- Establishes space management pages (SMPs) for the area(s) that map to the file
- Initializes the space management entry for each database page
- Establishes a header and footer on each database page
- Sets all data portions of database pages to binary zeros

Formatting Journal Files

When you issue a FORMAT command against a disk journal file, CA IDMS/DB formats the file into blocks according to the journal file specification in the DMCL module. The disk journal file contains:

- Journal header records at the beginning
- Binary zeros in the remainder

Before You Begin

Before you format a file, the DMCL that contains the file definition must be available. The DMCL provides the information CA IDMS/DB needs to format the file into database pages or journal file blocks.

Formatting Options

You can specify four options on the FORMAT utility statement. The following table identifies when to use these options:

Action	FORMAT Option
Format *newly*-created database file(s)	FILE or SEGMENT
Re-format non-empty database file(s) 1	AREA or SEGMENT*
Format a disk journal file	JOURNAL

Note: VSAM database files can only be formatted using the AREA option.

Example

The following example instructs CA IDMS/DB to format all the database files contained in segment EMPSEG:

```
format segment empseg;
```

Considerations for Native VSAM Files

About Native VSAM Files

A native VSAM file is a file that is already defined to VSAM and contains VSAM records. Even though a native VSAM file is not structured as a CA IDMS database file, users can gain access to it using CA IDMS/DB DML. To access data in native VSAM data sets, CA IDMS/DB converts DML statements issued by an application program into record-level (not control-interval) VSAM requests and passes control to VSAM. A CA IDMS/DB local run unit or the central version appears to VSAM as a single application that:

1. Opens VSAM data clusters
2. Activates VSAM paths using local-shared resources (LSR) or non-shared resources (NSR)
3. Accesses data records
4. Closes the clusters and paths

Native VSAM Files Contain Data

CA IDMS/DB can access native VSAM files only if they contain at least one record; that is, the files cannot be empty. This also implies that empty native VSAM files cannot be loaded using CA IDMS/DB services.

Defining Native VSAM to CA IDMS

Before an existing VSAM file can be accessed using CA IDMS/DB DML statements, both a logical and physical description must be provided using non-SQL schema and physical DDL statements.

Note: For more information about defining native VSAM files, see Appendix D, "Native VSAM Considerations".

More Information

- For more information about creating and formatting z/VM files, see *CA IDMS Installation and Maintenance Guide—z/VM*.
- For more information about loading files, see Chapter 22, "Loading a Non-SQL Defined Database" and Chapter 4, “Defining Segments, Files, and Areas” and Chapter 27, “Modifying Physical Database Definitions”.
- For more information about disk journal file definition and modification, see Chapter 5, “Defining, Generating, and Punching a DMCL” and Chapter 27, “Modifying Physical Database Definitions”.
- For more information about syntax for the FILE and DISK JOURNAL statements, see Chapter 7, “Physical Database DDL Statements”.
- For more information about loading files, see Chapter 22, “Loading a Non-SQL Defined Database” and Chapter 23, “Loading an SQL-Defined Database”
- For more information about IDCAMS, see the appropriate IBM publication.
- For more information about using native VSAM files, see the *CA IDMS Database Design Guide*.

Chapter 18: Buffer Management

This section contains the following topics:

Planning Database Buffers

Tradeoffs to Consider

Buffers use space in main memory, but reduce the amount of I/O performed on behalf of your applications. You want to choose the optimal buffer attributes to achieve a balance between storage resources and I/O.

Considerations for assigning values to these attributes appear next, beginning with a discussion on how many database buffers to define.

How Many Buffers Do You Need?

Multiple Buffers Allowed

As a general rule, one large buffer is often adequate for most processing situations. However, you may need to define more buffers to:

- Enhance database performance
- Optimize storage use

Separate Buffers to Enhance Performance

To enhance run-time performance, you can associate individual files with separate buffers. This reduces contention for buffer pages.

For example, you can assign a frequently-used index to a separate file and then assign the file to a separate buffer. Applications can access this index in its own buffer, while CA IDMS/DB uses other buffers to hold database pages.

Separate Buffers to Optimize Storage Use

The size of a buffer page must be as large as the largest database page that uses the buffer. Therefore, you can optimize storage use by assigning files that contain the same or similar block sizes to the same buffer.

How Many Pages Should a Buffer Contain?

Minimum Number of Pages

The minimum number of pages in a buffer is three. However, a value of at least five is recommended to avoid excessive database I/O operations and to reduce contention among transactions for space in the buffer.

Maximum Number of Pages

The maximum number of pages is constrained only by available memory resources. However, if you allocate *too many* pages, you may degrade performance by increasing the amount of virtual paging performed by the operating system.

Choosing an Optimum

Choosing an optimum number of pages comes with experience gained from tuning your database. However, if most files in the DMCL use a common buffer, a rule of thumb indicates that the number of buffer pages should be at least three times the maximum number of anticipated concurrent database transactions.

Manage the Size of the Buffer Dynamically in Response to Need

Once a database is in operation under the central version, you can dynamically change the number of pages in the central version buffer with a DCMT VARY BUFFER statement. By changing the size dynamically, you can determine the optimum size for the buffer by monitoring the *buffer utilization ratio*, which is described in 18.3, “Tuning Buffers for Performance".

Local Mode vs. Central Version Specifications

You can size a buffer differently for local mode and central version use. This feature allows you to optimize use of memory resources. For example, you could specify that a particular buffer will hold 100 pages when used in local mode and 500 pages when us ed under the central version. Under local mode, the buffer is smaller because it supports only a single application; under the central version, the buffer is larger because it supports multiple, concurrent applications.

Initial and Maximum Allocations Under the Central Version

Buffers defined to run under the central version can be assigned an initial number of pages and a maximum number of pages. Depending on the amount of system activity, you can use the DCMT VARY BUFFER command to change the number of pa ges in the buffer; for example, use the DCMT VARY BUFFER command to increase the number of buffer pages during peak system usage or to reduce the number of buffer pages at other times.

You Can Use JCL to Increment Size of Local Mode Buffer

At z/OS sites, you may want to increase the size of the buffer for a specific application, such as loading a database. You can do this without modifying the buffer definition by specifying additional buffer pages in the BUFNO parameter of the JCL statement identifying a file associated with the buffer. At runtime, CA IDMS/DB acquires storage for the buffer equal to the number of pages specified in the DMCL's buffer definition plus the value assigned to BUFNO for each file associated with the buffer.

Buffers and File Caching

In certain operating systems, you can cache database files in a separate storage area called a cache. There are two types of caching available:

- Memory caching in which files are cached in a dataspace or in z-storage (storage above the 64-bit address line)
- Shared caching in which files are cached in a coupling facility.

Note: For more information about how to enable file caching and the options available in different operating systems, see DMCL Statements.

If a file is cached, CA IDMS reads database pages from the cache into the database buffer. If it modifies the database page, CA IDMS writes the modified page back to disk and to the cache. One advantage of a cache is a reduction in the number of I/Os to the file. Another advantage is that you may be able to reduce the number of pages in your buffer pool, relying on the cache to hold pages while not in use.

Memory caching provides larger caching capabilities than database buffers (even those allocated above the 16-megabyte line). However, you must have sufficient expanded storage on your machine to support the use of memory caching. Without adequate storage, the paging overhead associated with the system can increase significantly.

If using a coupling facility cache, you must have enough coupling facility space to hold the most frequently accessed pages, to make its use worthwhile. An additional advantage of a coupling facility cache is that it can be shared by more than one central version.

File caching is not available for native VSAM files.

Using Batch LSR for VSAM Files

At z/OS sites, VSAM database files can make use of IBM's Batch Shared Resources Subsystem (Batch LSR) by specifying the SUBSYS JCL parameter. At runtime, CA IDMS/DB opens the VSAM database file and the VSAM Batch LSR subsystem converts the buffer management technique to LSR processing and allows the buffer pool to be created in hyperspace. Batch LSR is also supported for native VSAM files.

Batch LSR Improves Performance for Actively Used Files

By using Batch LSR, you can reduce the number of pages in the buffer associated with the file in your DMCL because VSAM and the Batch LSR subsystem can create a large buffer pool in hyperspace which will minimize the number of I/Os. This feature offers performance improvements for files that are actively used.

SUBSYS Subparameters

Use of the Batch LSR subsystem and the number and location of the buffers is controlled by use of the SUBSYS JCL parameter and its subparameters. Use the MSG=I subparameter to display the batch LSR subsystem messages on the job log. Do *not* use DEFERW=YES because it could affect the integrity of your database in the event of a system failure.

How Large Should a Buffer Page Be?

Pages as Large as Largest Database Page

The page size for a buffer must be able to hold the largest database page that will be read into that buffer. Therefore, to conserve system resources, try to assign files to the buffer with roughly equivalent block sizes (a block equals a database page).

Choosing a Method for Storage Acquisition

Choosing IDMS or OPSYS

The IDMS and OPSYS options on the BUFFER statements determine *how* CA IDMS/DB acquires storage for the buffer and the source of this storage:

- If you specify OPSYS storage, CA IDMS/DB issues one or more requests to the operating system for a contiguous block of storage. If the operating system supports extended addressing, the storage will be acquired above the 16-megabyte line.
- If you specify IDMS storage, CA IDMS/DB issues separate storage requests for each page in the buffer. The storage is acquired from IDMS-managed storage and will reside above the 16-megabyte line under the following conditions:
  - In local mode, if the operating system supports extended addressing
  - Under the central version, if an XA storage pool exists which supports system-type storage.

Advantages of Using OPSYS Storage

The OPSYS storage option offers an advantage to sites that define large buffers because of the way storage is acquired. For example, a buffer defined with an initial number of pages of 1000 will result in a single storage request for the entire 1000 pages if OPSYS is specified or 1000 storage requests if IDMS is specified. Another advantage is that the OPSYS storage is acquired outside the IDMS storage pool while IDMS storage is acquired from the IDMS storage pool. Therefore, the storage pool must be large enough to hold the buffer.

Insufficient Storage Under the Central Version

When initially allocating a buffer or when increasing the size of a buffer in response to a DCMT command, CA IDMS/DB may be unable to acquire all the necessary storage. If this occurs and the storage acquisition mode is OPSYS, CA IDMS/DB will attempt to acquire the storage from the IDMS storage pool. Whenever acquiring storage from the IDMS storage pool, if the necessary storage cannot be acquired or if the DC/UCF system is placed in a short-on-storage condition, the number of pages in the buffer is reduced by half until the necessary storage can be acquired without a short-on-storage condition.

Managing Buffers Dynamically

Changing Buffer Characteristics

Once a database is in operation, you can vary the characteristics of buffers dynamically by issuing the DCMT VARY BUFFER statement.

By making a temporary change to a buffer setting online, you can evaluate the potential impact this change might have on overall system performance. This allows you to identify the optimal settings for your buffers. When you have identified the optimal settings, you can make permanent changes to the buffer definitions by using the ALTER BUFFER DDL statement.

Types of Changes

The following buffer characteristics can be changed using DCMT commands:

- The number of pages in the buffer pool
- The number of pages to be acquired in each storage request (this value defaults to the initial number of pages in the buffer pool)
- The maximum number of pages in the buffer pool
- The storage acquisition mode (OPSYS or IDMS)
- Whether the chained read facility is activated and the number of pages that must be in the buffer to invoke chained reads as described under 18.4, “Using Chained Reads”
- Whether a file is cached using a DCMT VARY FILE/AREA/SEGMENT command

If the number of pages in the buffer pool is changed to any value between the initial and maximum number of pages, the change is effective immediately. Changing the number of pages in the buffer pool beyond this range or changing other buffer characteristics takes effect only after the buffer is closed and re-opened. The buffer can be closed using a DCMT VARY BUFFER command and it will be re-opened automatically when the next read occurs for a file associated with the buffer.

Varying a DMCL

The following buffer changes can be made dynamically by varying a new copy of the DMCL:

- The page size of a buffer can be changed
- New buffers can be added to the system
- Existing buffers can be removed from the system
- Files can be associated with a different buffer

Other characteristics, such as the number of pages in the buffer or the storage acquisition mode, are not affected by varying a new copy of the DMCL. To dynamically make such changes, use the DCMT VARY BUFFER command.

Tuning Buffers for Performance

When to Add More Database Buffers

If your monitoring operations reveal contention among applications for use of your buffers, you may need to add more buffers. For example, you may create a new buffer and assign it to a file that is accessed frequently; files that are accessed infrequently can share buffers without incurring contention among applications.

To determine which files within a buffer are accessed most frequently, issue the DCMT DISPLAY STATISTICS BUFFER command with the FILE option. This will show the number of pages requested as well as the number of reads and writes issued for each file associated with a specific buffer.

When to Change the Database Buffer Page Size

You may have to change the buffer's page size if you associate different files with the buffer. The buffer's page size must be as large as the largest database page in any file associated with the buffer. Therefore, if new files assigned to the buffer contain larger database pages, the buffer page must be increased accordingly; likewise, if the files are removed from the buffer, you may be able to decrease the buffer page size to conserve memory resources.

When to Change the Number of Database Buffer Pages

You can use the *buffer utilization ratio* to determine if a buffer has the optimal number of pages. This ratio is the number of database pages requested to the number of database pages CA IDMS/DB reads from disk. A high ratio (above 2) indicates an effective buffer size. A lower ratio indicates that the buffer has too few pages.

You can use the DCMT DISPLAY STATISTICS BUFFER command to determine these values. You can also obtain them from the Performance Monitor, JREPORTs, and SREPORTs.

Using Chained Reads

What Chained Reads Do

Chained reads allows CA IDMS/DB to read multiple blocks from disk with a single I/O request. It can significantly reduce both elapsed and CPU times for applications that process multiple contiguous pages within an area.

CA IDMS/DB automatically uses chained reads under z/OS and z/VSE both in local and central version processing under these conditions:

- Prefetch processing has not been disabled by issuing a DCMT VARY command or through a PREFETCH SYSIDMS parameter
- The file being accessed is non-VSAM
- The file is not cached
- The buffer pool for the file contains a page count of at least 255 pages

- And, one or more of the following applies:
  - An area sweep is being performed.
  - An SQL request is processed in such a way that multiple contiguous pages will likely be accessed (walking a clustered set or index, performing an index scan).
  - The number of pages in the buffer pool exceeds the pre-fetch limit. The default pre-fetch limit is 500 but this can be overridden using a DCMT VARY BUFFER command or through a PREFETCH_BUF SYSIDMS parameter.
  - One of the following utility functions is executing:
    - ARCHIVE LOG
    - BACKUP
    - BUILD INDEX
    - CLEANUP
    - MAINTAIN INDEX
    - PRINT LOG
    - PRINT SPACE
    - REORG
    - RESTRUCTURE
    - RESTRUCTURE CONNECT
    - UNLOAD
    - UPDATE STATISTICS
    - VALIDATE

    Note: Several other utilities such as ARCHIVE JOURNAL use QSAM processing for their sequential processing.

How Chained Reads Work

When chained reads is active, a single start I/O reads up to an entire track at one time. If some of the pages are already in core, those pages are skipped (that is, they are not read).

When IDMS/DB processes an entire area, it issues multiple start I/Os. Under the central version, without read drivers, two start I/Os will be issued; in local mode, as many as ten start I/Os will be issued (subject to buffer pool size). IDMS/DB overlaps multiple start I/Os to reduce elapsed time.

Controlling the Use of Chained Reads

Under the central version, use the PREFETCH option of the DCMT VARY DMCL, AREA, FILE, or BUFFER commands to control when to use chained reads. ON is the default unless it is overridden by a PREFETCH=OFF SYSIDMS parameter. OFF takes precedence over ON for subordinate entities. For example, varying PREFETCH OFF for an area will disable it for all files associated with that area.

The default prefetch limit of 500 pages can be overridden by specifying a PREFETCH_BUF SYSIDMS parameter. In central version, it can also be overridden by using the following command:

```
DCMT VARY BUFFER <buffer-name> PREFETCH <limit>
```

For example, if the limit for a buffer pool is set to 300, then (provided that there are at least 300 buffer pages) chained reads will be used for all access to files associated with the buffer unless it is explicitly disabled at the file, area, or system level.

Monitoring Effectiveness

To determine the effectiveness of chained reads in your system, use the OPER WATCH DB IO command, which displays the number of start I/Os and pages read using chained I/O for a given task.

It is possible that certain applications or processing loads may either experience no improvement or incur increased overhead because chained reads may cause pages to be prematurely flushed from the buffer. If such a situation occurs, you can disable chained reads for local mode or central version by specifying PREFETCH=OFF as a SYSIDMS parameter.

Using Read and Write Drivers

Read Drivers

A read driver performs "look-ahead" reads when CA IDMS/DB is instructed to sweep an area. When it is activated, it uses chained reads to read a track of pages beginning with the third or fourth tracks from the start of the area sweep and attempts to prefetch the pages that will be needed by the application. Use the DCMT VARY DB READ ON/OFF command to activate or de-activate the read driver for an area.

Write Drivers

A write driver facilitates writing pages from the buffer to disk. CA IDMS/DB invokes a write driver under these conditions:

- When a transaction is committed and the buffer contains at least five updated pages. The driver writes all the pages in the buffer updated by the transaction.
- When more than 75% of the pages in the buffer are updated pages.

Use the DCMT VARY DB WRITE DRIVER ON/OFF command to activate or de-activate the write driver.

More Information

- For more information about defining database buffers, see Chapter 5, “Defining, Generating, and Punching a DMCL”.
- For more information about caching files, see 7.13, “DMCL Statements".
- For more information about DCMT commands, see the *CA IDMS System Tasks and Operator Commands Guide*.
- For more information about shared cache, see the *CA IDMS System Operations Guide*.

Chapter 19: Journaling Procedures

This section contains the following topics:

Journaling Overview

Journals Log Database Activity:

Journals log database activity. Specifically, journals log:

- Before and after images of modified records and rows
- Status of transactions accessing the database

Note: Throughout the remainder of this chapter, the term record is used to mean both record and row

Journaling Under the Central Version

Update and Retrieval Transactions

Under the central version, several transactions can update the database concurrently. CA IDMS/DB writes information about all update transactions to the journal files. CA IDMS/DB also writes status information about retrieval-only transactions if JOURNAL RETRIEVAL is specified in the system generation SYSTEM statement.

Use Disk Journals Under the Central Version

You must use disk journals for automatic recovery under the central version. Automatic recovery occurs during *warmstart*, following the abnormal termination of a transaction, and following the execution of an SQL statement during which errors were encountered.

Note: For more information about automatic recovery, see 21.2, “Backup Procedures”.

Need at Least Two Disk Journals

Under central version, you need at least two disk journals. As one file becomes full, CA IDMS/DB automatically switches to an alternate file. While CA IDMS/DB writes to the alternate file, the full disk journal file must be offloaded using the ARCHIVE JOURNAL utility statement. This procedure is described in more detail later in this chapter.

Journaling in Local Mode

Journaling May Not Be Necessary

When you execute an application in local mode, that application is the only one that has access to any areas it updates. Therefore, journaling may not be necessary in local mode, provided you backup the database files before and after executing an application that updates the database. Typically, you journal in local mode when your database is too large to backup in a reasonable amount of time.

Must Use Tape Journals in Local Mode

To journal in local mode, you must use a DMCL that defines a tape journal file. You can assign the tape journal file to either a disk or tape device. However, if you journal to a disk device, you must copy the file to a tape device before using it in a manual recovery operation.

Journal Files

Journal Record Types

Database activity is recorded on a journal file (tape or disk). CA IDMS/DB writes the following information to the journal:

- Journal record entries that contain the image of database records
- Checkpoints that describe the status of transactions accessing the database

Writing Journal Blocks

CA IDMS/DB accumulates journal records in the journal buffer. It writes the journal buffer to a journal file when one of the following conditions occurs:

- The buffer is full.
- A page containing an updated record occurrence whose before image is in the journal buffer, is to be written back to the database.
- A database transaction is committed or backed out. A transaction commits or backs out as the result of an explicit request, such as a FINISH or ROLLBACK or because the application aborts.

Note: All journal file blocks are the same length, whether or not the buffer is full when written to a journal file.

Journal Record Entries

Log Changes in Records

CA IDMS/DB uses journal record entries to log changes to the records in a database. A journal record entry is an image of a database record. As a database record is added, deleted, or modified, CA IDMS/DB writes a *before image* that contains the image of the record before update and an *after image* that contains the image of the record after update.

Journal Images For Modified Records

On a change to an existing record, the contents of before and after images are dependent on how the processing of the DML statement affects the database record:

Affect on Database Record	Contents of Journal Record Entry
Data in the record changes	■ Database key of the record occurrence ■ Prefix portion of the record occurrence ■ Data portion of the record occurrence
Record's relationships in a set changes	■ Database key of the record occurrence ■ Prefix portion of the record occurrence

Journal Images For New or Deleted Records

If a DML statement adds a new record occurrence into the database, the *before* image of the record is null. Similarly, if a DML statement removes a record occurrence, the *after* image of the record is null.

Checkpoints

Describe Transaction Status

Checkpoints describe the status of database transactions. CA IDMS/DB writes these checkpoints to the journal file:

Checkpoint	Description
BGIN	Written to the journal file to mark the beginning of local work done by a transaction branch. This checkpoint is written when a database transaction is initiated if JOURNAL RETRIEVAL is specified in the system generation, or when the first update occurs, otherwise.
ENDJ	Written to the journal file during a commit operation to mark the successful completion of a transaction branch.
COMT	Written to the journal file during a commit operation to mark the successful completion of a transaction branch. A COMT is similar to an ENDJ checkpoint except that it enables work done after the commit to be recorded on the journal file using the same local identifier (LID).
ABRT	Written to the journal file during a backout operation to mark the abnormal completion of a transaction branch.

Checkpoint	Description
AREA	Written for each area readied by an explicit DML READY command or readied automatically by the DBMS.
RTSV	Written automatically to the journal file each time CA IDMS/DB encounters an error while executing an SQL or physical DDL statement that updated the database. During recovery, CA IDMS/DB rolls back to the journal record designated by the RTSV checkpoint record.
TIME	Written to a journal each time the journal's buffer is initialized. However, the time and date fields contain binary zeros until the journal buffer is written to the journal file.
BFOR	Written to a journal each time a record is updated and carries the image of that record before the change was made
AFTR	Written to a journal each time a record is updated and carries the image of that record after the change was made
CKPT	Written to a journal to mark the simultaneous successful completion of multiple branches of a local transaction. This checkpoint is used to coordinate the commit of a local transaction involving multiple branches.
USER	Written to a journal via the WRITE JOURNAL command issued by a user program
JSEG	Written to a journal at the beginning of each disk journal segment. This record identifies the transactions that were active when that journal segment was started.
DSEG	Written periodically to the journal to identify the transactions that are active at a given point in time.
DCOM	Written to the journal file during a two-phase commit operation to mark the successful completion of a distributed transaction. It is written to the coordinator's journal file at the start of the second phase of the commit operation and to a participant's journal file when it is informed that its changes should be committed.
DBAK	Written to the journal file during a two-phase commit operation to mark the abnormal completion of a distributed transaction. It is written to the coordinator's journal file during the first phase of the commit operation as soon as it is determined that the transaction's changes should be backed out and to a participant's journal file when it is informed that its changes should be backed out.

Checkpoint	Description
DFGT	Written to the journal file at the end of a two-phase commit operation to mark the end of the distributed transaction. It is written to a coordinator's or participant's journal file if any other distributed checkpoint (DCOM, DBAK, DPND, and DIND) had been previously written for the transaction.
DIND	Written to a participant's journal file during a two-phase commit operation to note that the participant is prepared to commit its portion of the distributed transaction. The participant will wait for a directive from the coordinator as to whether to complete the commit operation or to back out changes.
DPND	Written to the journal file during a two-phase commit operation to mark an interim result for a distributed transaction. It may be written for several reasons, such as when a participant is forced to complete a transaction heuristically or when a coordinator is unable to communicate with a participant during the second phase of the commit operation.

Note: ENDJ, COMT, and ABRT checkpoints are written to the journal file only by transactions for which a BGIN checkpoint is also written.

Avoiding Duplicate LID Values

An LID is a serially incremented 4-byte value that uniquely identifies work done by a local transaction branch. With the increased need for non-stop operations, there is a greater chance that LID values may wrap within the lifetime of a central version. This, in turn, could lead to a situation in which duplicate LID values are in use, if a database session existed long enough for its LID value to be repeated. To minimize this possibility, you can force the assignment of a new LID value each time a long running database session commits or backs out its work. Doing so has the added benefit of reducing recovery time.

To force the assignment of a new LID value, specify ON COMMIT WRITE ENDJ NEW ID and ON ROLLBACK NEW ID on the SYSTEM or TASK statements in the system definition or specify this dynamically using the DCMT VARY TASK or VARY DYNAMIC TASK commands.

Note: There is no need to specify this for system run units, since a new LID value is always assigned when they are committed or backed out.

More Information

- For more information about improving recovery times, see 19.6.3, "Reducing Recovery Time".
- For more information about specifying system definition parameters, see the *CA IDMS System Generation Guide*.
- For more information about DCMT commands, see the *CA IDMS System Tasks and Operator Commands Guide*.

Two-Phase Commit Journaling

Journaling and Two-Phase Commit

Journaling is integral to the two-phase commit process. As a distributed transaction progresses through the commit process, distributed checkpoint records are written to the journal file to record its changing state. These checkpoints are used in the event of a system failure to rebuild incomplete transactions so that they can be completed through a process of resynchronization with other systems.

Note: For a complete discussion of two-phase commit within CA IDMS, see Chapter 20, “Two-Phase Commit Processing".

Distributed Checkpoints

The following distributed checkpoints can be written in support of a two-phase commit operation:

- DIND (In doubt) - Written by a participant after it has successfully prepared its resources for commit and prior to returning an OK response to its coordinator.
- DCOM (Commit) - Written by a coordinator to signify that a transaction's changes will be committed. Its existence separates the first and second phases of the commit process. A participant also writes a DCOM immediately upon receiving a *Commit* request from its coordinator.
- DBAK (Backout) - Written by a coordinator to signify that a transaction's changes will be backed out. Its existence demarcates the first and second phases of the commit process. A participant also writes a DBAK immediately upon receiving a *Backout* request from its coordinator, but only if a DIND had been previously written.

- DPND (Pending) - Written by a coordinator during the second phase of a commit operation if some participant is unable to complete its commit processing due to a failure. By writing this record, the coordinator is able to forget some participants while remembering others. It is also written to record the heuristic completion of a transaction.
- DFGT (Forget) - Written by coordinators and participants when they have completed their two-phase commit processing for a transaction. A DFGT record is written only if some other distributed checkpoint record was previously written.

Each distributed checkpoint contains a distributed transaction identifier (DTRID), a 16-byte value assigned by CA IDMS to uniquely identify a distributed transaction across all participating systems.

Relating Local and Distributed Journal Entries

BGIN, COMT, ENDJ and ABRT checkpoints and BFOR and AFTR journal entries log work done by a transaction branch within the local system. They contain a 4-byte local identifier (LID) that uniquely identifies this work. To associate work done locally with a distributed transaction, DIND, DCOM and DBAK checkpoints contain a list of LID values representing the local work units that are part of the distributed work unit.

The following illustrates the sequence in which local and distributed journal records may be written to a journal file for a distributed transaction:

- BGIN - indicating the start of work done locally
- BFOR/AFTR - one or more pairs
- DIND - on a participant only
- DCOM or DBAK - on a participant and a coordinator
- COMT or ENDJ - if a DCOM was written
- ABRT - if a DBAK was written
- DPND - on a coordinator if the commit operation was interrupted; on a participant if the transaction was heuristically completed
- DFGT - on a participant and a coordinator if any other distributed checkpoint was written

Participants and Coordinators

DIND, DCOM, DBAK and DPND records also contain information about a participant's coordinator and about a coordinator's participants. The specific information that is recorded varies depending on the type of the coordinator or participant. For example, the node name, resource name, and remote transaction branch identifier are recorded for CA IDMS participants. The RRS URID (Unit of Recovery Identifier) is recorded for an RRS coordinator or participant.

Splitting Distributed Checkpoints

Distributed checkpoint records can be larger than a single disk journal block. If this is the case, they are split into as many journal blocks as are necessary to hold the entire record. It is also possible for a distributed checkpoint to be split across disk journal files and, hence, across archive files. The manual recovery utilities reassemble the record, provided that all necessary archive files are processed in a single execution of the utility. They ignore partial records in which not all segments are present in the input file.

I/O Error or Corruption of a Journal File

While CV is active, if a journal encounters an I/O error, or some type of corruption, IDMS will stop using the journal. If the error occurred to the active journal, IDMS will switch to the next journal, and disable the use of this journal. If the error occurred to a journal that was not the active journal, then IDMS will not switch to this journal. It will remain disabled to IDMS until the journal file is corrected.

If two journals are defined to the system, and one becomes corrupted, IDMS will journal to just one journal. When that journal fills, IDMS will wait until the journal is offloaded by the ARCHIVE JOURNAL, before it starts to journal again. All update activity to the database will stop until the ARCHIVE JOURNAL is done. Therefore, it is recommended that you define three or more journals to the system. Most users run with four or more journal files.

To correct the disabled journal; Run an Archive Journal - to offload any data on the corrupted journal. If the offload will not run, identify the journal file that is corrupt and resubmit the Archive Journal using the READ parameter.

DCMT VARY JOURNAL journal-file-name INACTIVE. This makes sure the journal is 'offline' or disabled to IDMS. IDMS will not use the journal at this time.

Format the journal. Run the IDMS FORMAT Utility, specifying just the journal file with the error.

DCMT VARY JOURNAL journal-file-name ACTIVE. This brings the journal back 'online' to IDMS. After this command is issued, IDMS will automatically switch to this journal when the prior journal is full. Until this journal is switched to, the Segment number for this journal will be '0'.

Formatting Journal Files

Before a journal file can be used by a system, it must be formatted. When doing so, it might be necessary to specify a size for the journal data store. This is an area reserved at the start of every journal file that is used to record information about other systems with which a system communicates. In most cases, the default size is sufficient and no explicit size parameter is needed; however, if a system's journal block size is very small, or if it communicates with many other CA IDMS or CICS systems, it may be necessary to reserve additional space. If a journal's available space is exhausted, attempts to communicate with a new system will fail. It will then be necessary to shut down the system, offload and format the journal files, and restart the system before communicating with a new system.

You specify a size for the journal data store by specifying a DATA STORAGE SIZE parameter on the FORMAT JOURNAL statement. All journal files used by a system must be formatted with the same data store size. Once a journal file has been formatted for the first time, it can be reformatted using the FAST option. This parameter directs the utility to format only the journal headers which is a much faster process than formatting the entire file.

For more information about formatting journal files, see the *CA IDMS Utilities Guide*.

Offloading Disk Journal Files

What Happens When You Offload a Disk Journal File

The ARCHIVE JOURNAL utility statement offloads the contents of a disk journal file to an archive journal file. It also rebuilds the disk journal file, condensing all before images for each active transaction into new journal blocks at the beginning of the file. This process creates a journal file that contains only those before images that are needed if an active transaction aborts or requests rollback.

Creating Multiple Archive Files

CA IDMS/DB will offload the disk journal files to multiple archive files if more than one is defined in the DMCL used when executing the ARCHIVE JOURNAL utility statement. By creating multiple archive files, you increase the likelihood that a readable archive file is available in the event it is needed for manual recovery. If an I/O error is encountered while writing to one of the archive files, a warning message is issued and offloading continues without further writes to the damaged file. If all archive files incur write errors, execution is aborted.

When to Offload

You normally offload disk journal files only when:

- CA IDMS/DB switches to another disk journal file
- The DC/UCF system is shut down and the database is backed up

The procedure for each scenario is provided next followed by a description of how to restart an offload operation.

When CA IDMS/DB Switches Journal Files

When Switch Occurs

CA IDMS/DB switches to another disk journal file when:

- The active disk journal becomes full
- You issue a DCMT VARY JOURNAL command under the central version
- An I/O error is detected on the active disk journal file

What Happens When the Switch Occurs

When CA IDMS/DB switches to another disk journal file, it writes a message to the operator, indicating that a swap has occurred and that the previously active journal file needs offloading. The operator should respond to this message by offloading the full file.

Eliminating Operator Intervention

You can eliminate the need for operator intervention by using a write-to-operator exit routine that intercepts and reviews the message to the operator and responds by automatically submitting a job to offload the full journal file.

Note: For information about the WTOEXIT user exit and sample routines for each operating system, see the *CA IDMS System Operations Guide*.

How to Offload the Disk Journal

ARCHIVE JOURNAL Utility Statement

To offload the journal, you execute the ARCHIVE JOURNAL utility statement using the batch command facility. You should use the default option of AUTO so that the oldest non-archived journal file is selected for processing.

If change tracking is in effect for the CV whose journals are being offloaded, you should reference the CV's SYSTRK file in your ARCHIVE JOURNAL execution jcl so that the archive job is sharing the description of the journal files that are currently in use by CV.

Note: For more information about change tracking and how to reference a SYSTRK file, see "Change Tracking" in the *CA IDMS System Operations Guide*.

System Failure During Offload

If the operating system fails while an ARCHIVE JOURNAL statement is executing, resubmit the ARCHIVE JOURNAL job using the RESTART parameter and identifying the journal file that was being processed at the time of failure.

Potential Problems While Offloading

You may encounter two types of problems when you offload journal files in an active system:

1. The offloaded journal file is still full following the offload because it contains before images for uncommitted transactions active at the time of the offload. The ARCHIVE JOURNAL utility statement issues messages indicating how full the disk journal file is after being offloaded. If it is full, it is usually because a long-running batch job is updating the database without issuing intermediate COMMIT statements. If all journal files fill, the system will halt database processing until corrective action is taken. See 19.4.3, “Handling Full Journal Files" for how to recover from this situation.
2. The remaining disk journal files fill before ARCHIVE JOURNAL completes offloading a single file. When this occurs, CA IDMS/DB temporarily halts further database activity until the offload job is complete.

Prevention For Problem 1

To prevent a full disk journal following an offload, take one or more of the following steps:

- Ensure that batch update programs issue frequent COMMITs to reduce the number of before images that must be retained on the journal file
- Allocate larger disk journal files
- Execute long-running update programs in local mode

Prevention For Problem 2

To prevent future disk journal file overloading, take one or more of the following steps:

- Allocate larger disk journal files
- Increase the number of disk journal files
- Execute long-running update programs in local mode.

Handling Full Journal Files

Long running transactions that do not commit their changes can fill the journals because the ARCHIVE JOURNAL utility is unable to remove the BFOR images for uncommitted transactions. The ARCHIVE JOURNAL utility generates a warning message when the journal file that is being archived remains nearly full after the process completes. The ARCHIVE_JOURNAL_WARNING_PERCENT SYSIDMS parameter dictates the threshold used to trigger the message. If this message appears, it indicates that the journal files are filling and corrective action may be needed.

When the journals are close to being full, the CA IDMS system halts database activity. To recover from this situation, the task that is filling the journals must be canceled.

To assist in this process, the following message is written for each task that is waiting to write to a full journal file:

```
DC205024 Journal Write waiting on full Journal
```

The message will be repeated every few seconds until tasks are no longer waiting on a full journal.

To recover from this situation, the task that is filling the journal files must be identified and aborted. To assist in this effort, CA IDMS will display message DC205030 showing details of the task causing the journal swap. This is most likely the task causing excessive journaling activity, although it may not be. To determine if this is the offending task, display detailed transaction information by issuing DCMT DISPLAY TRANSACTION commands or using the transaction detail function of the real time monitor within CA IDMS Performance Monitor.

Look for the transaction that has written the largest number of BFOR journal images. Cancel its associated task by issuing a DCMT VARY ACTIVE TASK command.

When the cancelled task has terminated, issue a DCMT VARY JOURNAL command to force the central version to swap to a new active journal file allowing the full file to be offloaded and condensed by ARCHIVE JOURNAL. It is likely that DCMT VARY JOURNAL will need to be issued more than once, since several journal files may have filled and require offloading.

Once the system swaps back to the first journal file on which tasks waited, processing should continue without the need for further intervention.

After System Shutdown

Offload All Files

After a normal system shutdown, you may offload *all* non-empty journal files, by executing an ARCHIVE JOURNAL utility statement with the ALL option:

```
archive journal all;
```

Usually Done in Conjunction With Backup

Offloading all journal files following a system shutdown is usually performed in conjunction with backing up the database.

Note: For more information about backup, see 21.2, “Backup Procedures".

User Exits and Reports for Journal Management

User exits

The following table describes user exits that you can use in managing your journals:

User Exit	Description
IDMSAJNX	Can be used to collect statistics on database activities; CA IDMS/DB invokes this exit as it offloads a journal record page to the archive file
IDMSDPLX	Can be used to maintain duplicate journal files; CA IDMS/DB invokes this exit each time it writes to the disk journal or a database file;
IDMSJNL2	Can be used for duplicating journal information and statistics collection; CA IDMS/DB invokes this exit each time it writes a journal buffer to the journal file
WTOEXIT	Can be used to automatically initiate a journal offload following a switch to a new journal file. CA IDMS/DB invokes the exit each time a message is written to the operator.

Note: For more information about these user exits and how to invoke them, see the *CA IDMS System Operations Guide*.

Reports

The following table summarizes reports you can use to manage your journals:

Reports	Description
JREPORTs	Report on the content of the journal file as follows: ■ Transaction summary ■ Program termination statistics ■ Program I/O statistics ■ Program summary ■ Transactions within an area ■ Programs within an area ■ Area summary You can also request a formatted dump of the journal file
PRINT JOURNAL utility	Reports on checkpoint information for transactions recorded on the archive file; this information is useful for rollback and rollforward operations

Note:

- For more information about JREPORTs, see the *CA IDMS Reports Guide*.
- For more information about the PRINT JOURNAL utility, see the *CA IDMS Utilities Guide*.

Influencing Journaling Performance

CA IDMS/DB provides facilities to:

- Reduce the amount of I/O activity for journal files under the central version
- Reduce the time needed to warmstart a central version following abnormal termination
- Reduce the time needed to recover long running database sessions

Reducing Journal File I/O

Increasing Journal Buffer Size

If your system encounters frequent or sizable rollback operations, it may be possible to reduce the I/O to the journal file by increasing the number of pages in the journal buffer. Minimally, the journal buffer should hold at least 5 pages. Increasing the number of pages may significantly improve performance.

Deferring Journal Writes

You can reduce the amount of journal I/O by instructing CA IDMS/DB to defer the writing of journal buffers. Normally CA IDMS/DB forces the writing of a journal buffer to the journal file whenever a COMT, ENDJ, or ABRT record is written to the journal buffer or when an updated database page is written to disk and the journal buffer contains a before image for a record on that page. You can request that CA IDMS/DB defer the journal write by specifying a non-zero JOURNAL TRANSACTION LEVEL either in the system generation SYSTEM statement or in a DCMT VARY JOURNAL command.

How Transaction Levels Work

When the number of active transactions in the central version is greater than the journal transaction level, CA IDMS/DB defers the writing of a journal buffer. If the journal write is deferred, the task requiring the write is placed in a wait state until the journal block is written. The journal block is written when:

- The number of active transactions falls below the journal transaction level
- The journal buffer is full

Note: An 'active transaction' is one for which journal records are being created.

By deferring the journal write, CA IDMS/DB is able to place more information on a journal block, thus reducing the need to write as many blocks.

Considerations

The establishment of a journal transaction level is most effective in an active system; that is, one in which many update transactions are active at one time. If used, you should set the journal transaction level to be at least 4. The lower the number, the longer tasks deferring their journal writes may wait.

Improving Warmstart Performance

Reducing Warmstart Time

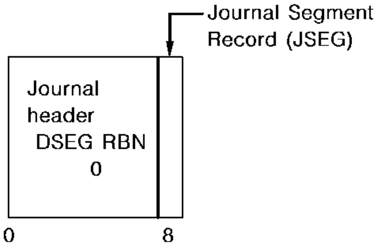
You can reduce the time it takes to warmstart a central version following an abnormal termination by specifying a non-zero value for a JOURNAL FRAGMENT INTERVAL in the system generation SYSTEM statement or in a DCMT VARY JOURNAL command.

How the Journal Fragment Works

The journal fragment interval designates an interval for writing dummy segment (DSEG) records to the journal file. DC/UCF uses the DSEG records in the event of a system crash to determine the appropriate starting place for warmstart processing, as shown in the following steps:

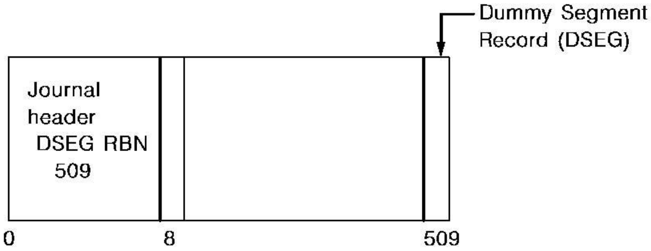
1. The new journal file is activated. It begins with header records. These records contain:

   - Information on currently open transactions
   - The relative block number (RBN) of the DSEG record. The RBN signifies which DSEG record is used to start forward processing in the event of a warmstart.

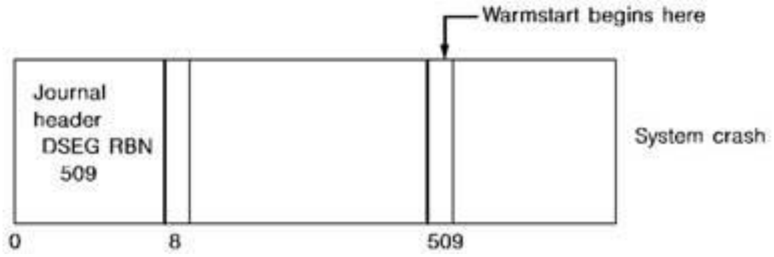


2. If the journal fragment interval is 500, the DC/UCF system will do the following before it writes the 509th journal block:

   - Creates and writes the DSEG record
   - Updates the DSEG RBN in the journal header



3. In the event of a system crash, the warmstart forward processing starts at the DSEG record at RBN 509 instead of at the JSEG record. This saves the time it would have taken for processing to read the first 500 journal blocks.



Considerations

If your journal files are large (in terms of the number of pages), a journal fragment interval can significantly reduce the amount of time it takes to warmstart a DC/UCF system. The warmstart logic goes to the most recently accessed journal fragment and starts its recovery processing from that point. However, because there is overhead required to write dummy segment headers, your journal fragment interval should be at least 100. Choose an interval that is between 100 and half the number of blocks in your journal file.

Reducing Recovery Time

Another way to reduce recovery time for all types of recovery operations is to force the writing of an ENDJ checkpoint instead of a COMT for long-running database sessions that periodically commit their changes. This is especially useful for long running sessions that infrequently perform a burst of updates and then issue a commit. Forcing an ENDJ reduces recovery time because less data has to be examined to locate the start of a recovery unit. This can benefit all types of recovery: warmstart, automatic recovery, and manual recovery.

To force the writing of an ENDJ during commit operations, specify ON COMMIT WRITE ENDJ on the SYSTEM or TASK statement in the system definition or specify this dynamically using the DCMT VARY TASK or DCMT VARY DYNAMIC TASK commands.

Note: There is no need to specify this for system run units, since an ENDJ checkpoint is always written when they are committed.

More Information

- For more information about defining and modifying journal files, see Chapter 5, “Defining, Generating, and Punching a DMCL”and Chapter 27, “Modifying Physical Database Definitions”.
- For more information about database backup and recovery, see 21.2, “Backup Procedures”.
- For more information about allocating and formatting disk journal files, see Chapter 17, “Allocating and Formatting Files”.
- For more information about user exits, see the *CA IDMS System Operations Guide*.
- For more information about and the complete syntax and syntax rules for the ARCHIVE JOURNAL utility statement, see the *CA IDMS Utilities Guide*.
- For more information about DCMT VARY JOURNAL and DCMT VARY FILE commands, see the *CA IDMS System Tasks and Operator Commands Guide*.
- For more information about journal system generation parameters, see the SYSTEM statement in the *CA IDMS System Generation Guide*.

Chapter 20: Two-Phase Commit Processing

This section contains the following topics:

Two-Phase Commit Overview

Two-phase commit is a protocol used to ensure that all changes made within the scope of a distributed unit of recovery are either applied (committed) or backed out. As the name implies, a two-phase commit process is divided into two phases. In the first phase, resource managers participating in the unit of recovery prepare their resources to be committed. If they cannot do so, they inform the request or of the failure. In the second phase, the resource managers either make their changes permanent or back them out based on the overall outcome of the transaction.

If a resource manager indicates that it has successfully prepared its resources to be committed, it guarantees that the resources can be committed even if some adverse condition, such as a system failure, occurs prior to completion of the commit process. It is this guarantee that ensures that all changes are either applied or backed out in their entirety.

The remainder of this section first introduces some terminology related to two-phase commit processing and then describes some of the key aspects of a two-phase commit operation.

Terminology

The following terms are associated with two-phase commit processing:

A *Resource Manager* is a software component that controls access to and the state of one or more recoverable resources. A CA IDMS central version is an example of a resource manager.

A *Transaction Manager* is a software component that directs commit and backout processes. Multiple transaction managers may be involved in a single commit or backout operation. If so, their actions are coordinated to achieve transaction consistency. Every CA IDMS system has a transaction manager as a component.

A *Coordinator* is a transaction manager that initiates a two-phase commit operation and is responsible for its overall outcome. A coordinator is sometimes referred to as an initiator.

A *Participant* is a resource manager or a transaction manager other than the coordinator that participates in a two-phase commit operation. A participant is sometimes referred to as an agent.

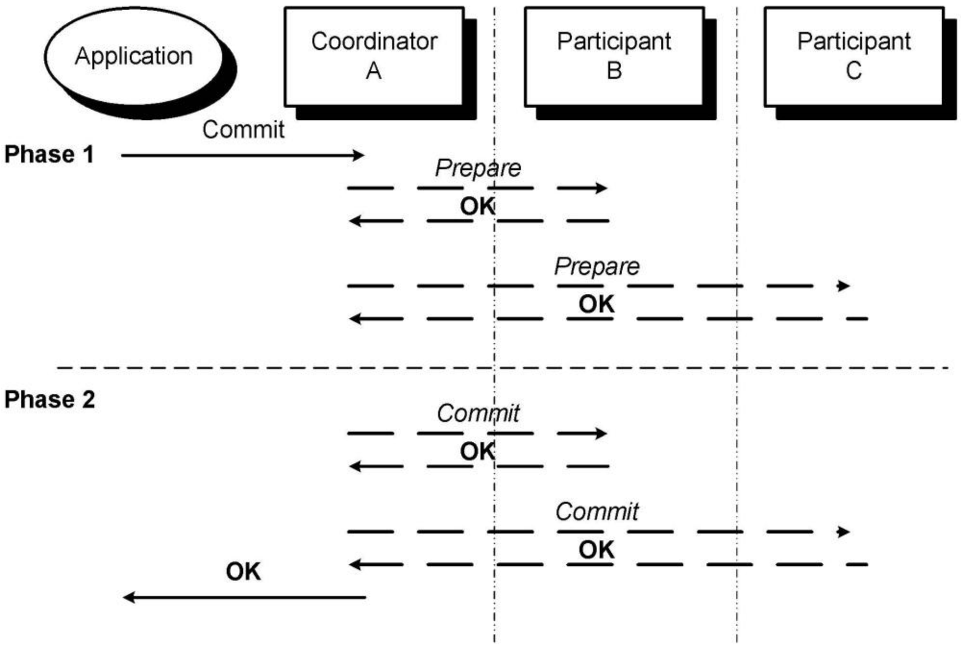
A *Distributed Transaction* is a unit of recovery in which more than one resource manager participate.

Typical Commit Flows

The following diagram illustrates the communications that take place during a typical two-phase commit operation involving three systems. In this example, A is the coordinator since it initiates the commit operation, and B and C are participants.

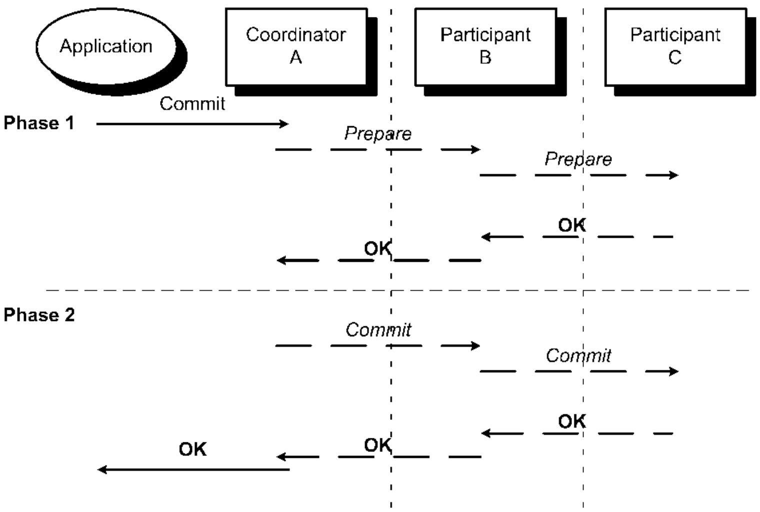
In this example, A forwards a *Prepare* request to each of its participants, directing them to prepare for a commit. After B and C both respond positively, A then directs them to complete the commit operation.

Commit Flow



The following diagram illustrates another typical commit flow. In this example, A is again the overall transaction coordinator, and B and C are participants. However, in this case, B plays a dual role. It is both a participant with respect to A and a coordinator with respect to C since it forwards the *Prepare* and *Commit* directives that it receives from A to C. Such a situation might arise because an application on A starts a remote SQL session on B that, in turn, updates resources on C through an SQL procedure.

Commit Flow



Prepare and Commit Outcomes

When a participant receives a *Prepare* request, it does whatever is necessary to guarantee that a subsequent *Commit* request can be honored. This may involve such things as flushing buffers or forwarding requests to other participants. If all of these activities are completed successfully, the participant signals its willingness to commit by responding OK to the *Prepare* request. If it is unable to successfully complete its preparations, it indicates this by responding BACKOUT to the *Prepare* request.

The coordinator gathers the responses from its participants and determines the final outcome for the commit operation. If all participants indicate that they are willing to commit, then the coordinator proceeds with the second phase and the transaction will complete successfully as indicated by a final outcome of OK. If any participant indicates that it cannot commit, then the coordinator directs its participants to back out their changes instead of committing them. The final commit outcome in this case is BACKOUT.

A participant can respond to a *Prepare* request in ways other than OK or BACKOUT. It can respond FORGET to signal that it made no updates within the transaction being committed and so need not participate in the second phase. This has the potential for reducing the number of communications needed to complete the commit operation. A participant can also respond "heuristically", indicating that its resources have already been committed or backed out. A transaction might be completed heuristically because it was forced to complete through some administrative action. Such heuristic actions defeat the two-phase commit process and can lead to mixed outcomes in which some changes are committed while others are backed out.

Note: While CA IDMS does not make heuristic decisions on its own, it does allow an administrator to commit or backout a transaction using a DCMT command, following an interruption in the commit process.

Recovery From Failure

Failures in communications, operating systems, or resource or transaction managers can interrupt the two-phase commit process. The point at which the failure occurs determines whether a transaction's changes are ultimately committed or backed out. If the failure occurs during the first phase in the process, changes are backed out. If the failure occurs during the second phase, changes are committed.

Recovery from failure during a two-phase commit involves a process called resynchronization, in which messages are exchanged between a coordinator and a participant in order to complete the transaction. To facilitate resynchronization, both the coordinator and the participant write additional journal records at critical points during the two-phase commit process.

Two-Phase Commit within CA IDMS

This section describes the two-phase commit support provided by CA IDMS. While some of the information may not be necessary for day-to-day operations, it facilitates understanding the output from recovery utilities and DCMT commands and may prove useful in recovery situations.

Use of Two-Phase Commit

Two-Phase in Central Version

Central version always uses a two-phase commit protocol to commit resources. When a commit operation initiates within a central version, that system becomes the coordinator. Any other central version involved in the transaction becomes a participant.

Requesting Two-Phase Commit

The DML commands that an application issues to commit tasks and database transactions (for example, FINISH TASK or COMMIT CONTINUE) always initiate a two phase-commit. This ensures that all changes made even by a distributed transaction are either committed or backed out as a single unit. A number of optimizations are supported to minimize overhead, especially for transactions in which only a single resource manager has made changes. See "Commit Optimizations" later in this section for a description of the optimizations supported by CA IDMS.

Support for Pre-Release 16.0 Central Versions

Prior to Release 16.0, CA IDMS did not support a two-phase commit protocol. Such a system is referred to as a "one-phase commit only" resource manager, since it can accept only a single commit request rather than separate *Prepare* and *Commit* requests. CA IDMS supports one-phase commit only resource managers in the following way.

If there is a single one-phase commit only participant in a distributed transaction, CA IDMS first sends *Prepare* requests to all other participants before sending a commit request to the one-phase commit only participant. If this latter request is successful, then the commit operation proceeds to a successful conclusion; otherwise, the transaction is backed out.

If a distributed transaction has multiple one-phase commit only participants, only one of them can be the last one invoked and so there is an unavoidable possibility that the transaction may complete with mixed results, meaning that some changes are committed while others are backed out.

Support for Batch Applications

All changes made by a batch application are committed or backed out as a single unit provided at least one of the following is true:

- All updates are made through a single transaction
- Updates are made through multiple transactions serviced by a single central version and a task-level commit request is issued.
- Batch RRS support is enabled and all updates are made through transactions executing on one or more central versions running within the same operating system image as the batch application.

Note: For more information about RRS, see the *CA IDMS System Operations Guide*.

External Coordinators and Participants

External Coordinators

A central version can participate in a two-phase commit operation controlled by the following external coordinators:

- CICS Transaction Server
- RRS-IBM's system-level resource recovery platform for z/OS
- An XA transaction manager supported by CA IDMS Server

By participating in externally-controlled transactions, updates to CA IDMS resources can safely be coordinated with those of other resource managers supported by the above transaction managers.

Note: For more information about CICS Transaction Server and RRS, see the *CA IDMS System Operations Guide*.

External Participants

A central version can coordinate transactions in which external resource managers are participants. It does this in one of two ways: by enlisting the services of RRS or by using a resource manager interface tailored to both the CA IDMS environment and the external resource manager.

If the external resource manager supports RRS as a coordinator, using RRS as an intermediary is the easiest way for an external resource manager to participate in a transaction coordinated by a central version. In this way, any resource manager that supports RRS as a coordinator can potentially participate in a CA IDMS-controlled transaction.

If the resource manager does not support RRS as a coordinator, then an interface that is tailored to the external resource manager and that supports the CA IDMS transaction manager protocol must be written to enable the resource manager to be a direct participant in a CA IDMS-controlled transaction.

Resource Managers, Interfaces and Exits

Resource Managers

When discussing commit protocols, the term "resource manager" traditionally refers to a software component that manages recoverable resources. However, in CA IDMS this term refers to both the resource manager and the interface used to communicate with it.

The DCMT DISPLAY DISTRIBUTED RESOURCE MANAGER command can be used to obtain a list of all resource managers known to a central version or to display the details of a specific resource manager.

Note: For more information about DCMT commands, see the *CA IDMS System Tasks and Operator Commands Guide*.

CA IDMS Resource Managers

When one CA IDMS system accesses another, each one becomes a resource manager to the other. On the front end, the partner system is identified by its node name and an interface name of "DSI_CLI"; on the back end, the partner system is identified by its node name and an interface name of "DSI_SRV". A central version may have knowledge of several resource managers whose interface name is DSI_CLI or DSI_SRV, since it may communicate with several CA IDMS systems. Furthermore, a central version may have knowledge of two resource managers with the same node name, one for each of the two interfaces, since a system can act as both front-end and back-end to another system.

External Resource Managers

When a CICS system is used to access a central version, it becomes a known resource manager on the backend with a node name that is the concatenation of "CICS" and the CICS system name and an interface name that is specified by the IDMSINTC interface module through which access is made.

When RRS is activated within a central version, it is known as a resource manager whose node name is that of the local system and whose interface name is "RRS_RMI".

Note: For more information about CICS and RRS, see the *CA IDMS System Operations Guide*.

Interfaces and Exits

To participate in a two-phase commit operation coordinated by CA IDMS, a resource manager makes its existence known by registering with the local transaction manager. When registering, the resource manager interface identifies exit routines to be invoked by the transaction manager during the commit process. In this way, the resource manager interface acts as a bridge between the local transaction manager and the resource or transaction manager to which it provides access. It is the resource manager interface's responsibility to forward *Prepare*, *Commit*, and *Backout* directives and return appropriate responses to the local transaction manager.

When a resource manager's exit is invoked, it returns outcomes that are similar to those outlined in 20.2.8, “Transaction Outcomes". For example, a resource manager's prepare exit can return a FORGET outcome to signify that it has made no changes within the scope of the transaction and therefore need not participate in the second phase of the commit operation.

Interests and Roles

Transaction Interests

For a resource manager to participate in a transaction, it must register an *interest* in that transaction. The existence of an interest informs the transaction manager that the resource manager's exits should be invoked during commit and backout processing for the transaction.

Roles

When an interest is registered, the *role* that the resource manager is to play with respect to the transaction is specified. CA IDMS recognizes the following roles:

- *Communications Resource Manager (CRM)* - indicating that the resource manager is a remote participant in the transaction.
- *Participant (PART)* - indicating that the resource manager is a local participant in the transaction.
- *Server Distributed Resource Manager (SDSRM)* - indicating that the resource manager is the coordinator for the transaction.

When a central version application calls a resource manager interface to access a remote resource, the interface registers a CRM interest in the application's current transaction, since it is acting as a remote participant in that transaction. An SDSRM interest is registered before or during the prepare stage of transaction processing to record a remote transaction manager as the coordinator for a transaction.

As a two-phase commit operation proceeds, interests are assigned states similar to those outlined in 20.2.7, "Transaction States". For example, if an interest's prepare exit returns OK, then the state of the interest is set to InDoubt, reflecting the fact that the associated resource manager is waiting for the final commit or backout directive.

Commit Optimizations

Types of Optimizations

To minimize the cost of doing a two-phase commit operation, CA IDMS supports the Read Only, Single Agent, and Presumed Abort optimizations.

Read Only

The *Read Only* optimization reduces the number of communications needed to commit a distributed transaction. A participant that has not updated resources within the scope of the transaction can respond FORGET to a *Prepare* request. CA IDMS does not include such read-only participants in the second phase of the commit operation, thus eliminating at least one communication. Additionally, the read-only participant writes no journal records in support of the two-phase commit operation.

Single Agent

CA IDMS uses the *Single Agent* optimization to reduce the flows needed to commit a distributed transaction. At the point when a *Prepare* request is to be sent to the last remaining participant, if all other participants have responded FORGET or if this is the only participant in the transaction, then a *OnePhaseCommit* request is sent instead of a *Prepare*. This results in only a single communication with the participant to complete the commit operation. Furthermore, if there is only a single participant, the coordinator writes no journal records in support of the distributed transaction.

Presumed Abort

CA IDMS uses a *Presumed Abort* protocol to reduce journaling overhead. Simply put, this means that while a coordinator retains knowledge of a committed transaction until all of its participants indicate that they have completed the second phase of the commit operation, the coordinator can immediately forget transactions whose outcome is BACKOUT. Consequently, no journaling activity for a distributed transaction takes place at a coordinator until all *Prepare* votes have been collected and then only if the outcome is OK. The absence of knowledge of a transaction signifies that its outcome is BACKOUT.

The alternative to Presumed Abort is *Presumed Nothing*. Under this protocol, a coordinator retains knowledge of the outcome of a commit operation until all participants indicate that it can be forgotten, regardless of whether the final outcome is OK or BACKOUT. Consequently, a coordinator must journal the existence of a transaction prior to forwarding the first *Prepare* request, and it must retain knowledge of backed out transactions longer. CA IDMS does not support the Presume Nothing protocol.

Transaction Identifiers

Multiple Identifiers

Transactions can have multiple identifiers. CA IDMS assigns two types of identifiers: a local identifier and a distributed transaction identifier. External transaction managers may assign transaction identifiers of their own, generically referred to as external transaction identifiers.

Local Identifier

A *Local (transaction) Identifier (LID)* is a four-byte value that identifies the work done by a local transaction branch. It is used to distinguish the work done by one branch from that of another within a central version and is recorded in the journal records that are used to track local database changes (for example, BGIN, BFOR, AFTR). Local transaction identifiers are unique only within a central version.

Distributed Transaction Identifier

A *Distributed Transaction Identifier (DTRID)* is a 16-byte value that uniquely identifies a distributed transaction across all participating nodes. It is assigned by the CA IDMS system that is acting as the coordinator for the transaction or by a CICS interface. Every distributed transaction processed by a CA IDMS system is assigned a DTRID, regardless of whether the transaction also has externally assigned identifiers. The DTRID is recorded in the distributed transaction journal records that are written during the two-phase commit process (for example, DIND, DCOM, DFGT).

A DTRID value is comprised of an 8-character prefix followed by an 8-byte hexadecimal value. If assigned by a CA IDMS system, the prefix is the system's node name and the suffix is an 8-byte internal format timestamp. If the DTRID is assigned by a CICS interface, the 8-character prefix consists of "CICS" concatenated with the 4-character CICS system name. The 8-byte hexadecimal value is the UOW (Unit of Work) identifier assigned by CICS to the work unit being committed.

External Identifiers

External transaction managers may also assign their own identifiers to a distributed transaction in which CA IDMS is a participant. The following types of external identifiers are recognized by CA IDMS and are recorded in the distributed transaction journal records written by the central version that interfaces directly with the external transaction manager. These journal entries provide a cross reference between internal and external identifiers.

- **RRS URID** - the Unit of Recovery (URID) assigned by RRS. A URID is a 16-byte hexadecimal value.
- **XA XID** - the transaction identifier assigned by an XA transaction manager. An XID is a hexadecimal value which can be up to 140 bytes long.

Transaction Branch

A *Transaction Branch* represents a separately identifiable portion of a transaction within which deadlocks cannot occur. Unless transaction sharing is in effect, every database session (every run unit or SQL database session) is associated with a separate transaction branch. When transaction sharing is in effect, multiple database sessions may share a single transaction branch. In so doing, they avoid deadlocking amongst themselves, since deadlocks are not possible for work performed under a single transaction branch.

An application is associated with multiple transaction branches if it opens concurrent non-sharing database sessions. Multiple branches can also result from the use of system services that access a dictionary, such as loading from a load area or accessing a queue area. If more than one transaction branch exists, they are organized hierarchically, meaning that there is a single top-level branch and one or more subordinate branches. The top-level branch represents either the work done by a single database session or all work done by a task. A subordinate branch always represents the work done by a database session or multiple sessions if transaction sharing is in effect. A subordinate branch may, in turn, have subordinate branches of its own, perhaps as a result of an SQL routine that opens its own database session.

Note: For more information about the relationship between database sessions and transactions and the use of transaction sharing to avoid deadlocks, see either *CA IDMS Navigational DML Programming Guide* or *CA IDMS SQL Programming Guide*.

Branch Identifiers

Every transaction branch is assigned a unique identifier that never changes. This *Branch Identifier (BID)* is an eight-byte hexadecimal value that is sometimes qualified by the node name of the local system to make it a globally unique value. A BID is different from an LID, since an LID is assigned each time a transaction is started, whereas a BID is assigned only when a branch is created. If multiple transactions are serially associated with a transaction branch, because an associated database session commits its work without terminating, then the branch's LID value will change, but its BID will not.

A commit operation is always targeted to a single transaction branch and encompasses all of its subordinate branches. The target branch becomes the *top-level branch* of the transaction and its subordinates become the *subordinate branches* of the transaction. If a task-level commit operation is initiated, the target branch is always the top-level branch in the task's hierarchy. If a database session-level commit operation is initiated, the target branch is the one associated with the database session through which the commit request is issued.

It is possible that more than one set of Prepare/Commit flows are sent to a single participant for a transaction, each directed to a different target branch. The BID of the target branch is carried in the associated distributed journal records, in order to distinguish one set from another. The target branch is also included in several messages that may be generated during a two-phase commit operation.

A DCMT DISPLAY DISTRIBUTED TRANSACTION command for a specific transaction lists all of the local branches associated with a distributed transaction. For a description of this command, see the *CA IDMS System Tasks and Operator Commands Guide*.

Transaction States

Transaction State

Transaction state is an attribute of a distributed transaction that reflects its progress through a two-phase commit operation. The CA IDMS transaction manager assigns the following transaction states for this purpose:

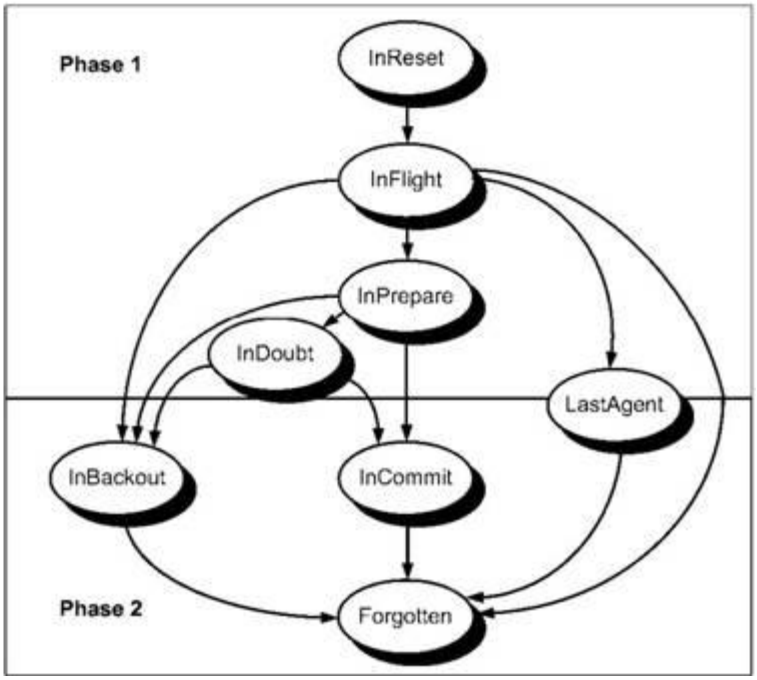
- InReset - This is the initial state prior to the start of a commit or backout operation.
- InFlight - This state is assigned at the start of a two-phase commit operation and persists while the transaction manager is assessing the need for and the ability to proceed with the two-phase commit operation.
- InPrepare - This state is assigned when the transaction manager determines that a full two-phase protocol is needed to guarantee the integrity of a commit operation.

- LastAgent - This state is assigned by a coordinator's transaction manager when there is only a single participant and consequently a full two-phase protocol is not needed to guarantee the integrity of a commit operation.
- InDoubt - This state is assigned by a participant's transaction manager when it writes a DIND journal record for the transaction and persists until it receives a *Commit* or *Backout* directive from the coordinator.
- InCommit - This state is assigned when a DCOM journal record is written for the transaction and persists until all processing for the transaction is complete.
- InBackout - This state is assigned when it is determined that the outcome of the distributed transaction is BACKOUT and persists until all processing for the transaction is complete.
- Forgotten - This state is assigned when the two-phase commit operation is complete.

State Transitions

The following diagram illustrates the transitions that can occur from one state to another as a transaction proceeds through a two-phase commit operation.

Transaction States



Transaction Outcomes

Transaction Outcomes

Fundamentally, a distributed transaction can have only one of the following three outcomes: all changes were committed, all changes were backed out, or some changes were committed while others were backed out. However, it is useful to support variations of these especially as interim results of individual *Prepare* and *Commit* requests.

CA IDMS recognizes the following outcomes:

- OK - The request is complete and the transaction's changes have been committed.
- FORGET - The request is complete, but no changes were committed since none were made (that is, this is a read-only transaction).
- OK_PENDING - The request is not yet complete, but changes have been or will be committed.
- BACKOUT - The request is complete but changes have been backed out.
- BACKOUT_PENDING - The request is not yet complete, but changes have been or will be backed out.
- HC - The request is complete, and the transaction's changes have been heuristically committed.
- HR - The request is complete, but the transaction's changes have been heuristically backed out.
- HM - The request is complete, but some changes have been committed while others have been backed out.

Heuristic Outcomes

Heuristic outcomes are the result of a commit or backout decision made by a participant rather than a coordinator. The use of DCMT VARY DISTRIBUTED TRANSACTION to force an InDoubt transaction to commit or back out results in a heuristic outcome.

Chapter 21: Backup and Recovery

This section contains the following topics:

Database Backup and Recovery Overview

Protects Your Data

Database backup and recovery are maintenance tasks that protect the changes made to your database:

- **Backup** is a routine database maintenance task that produces a copy of the database. If necessary, this backup copy can be used to restore lost data.
- **Recovery** restores the contents of the database when an error occurs that corrupts the database or disk journal file. Recovery procedures restore altered areas to their original state.

Types of Recovery

Under the central version, recovery occurs *automatically* with no intervention from the DBA. If automatic recovery fails you must recover the database *manually*. You must also recover the database manually for local mode update jobs that terminate abnormally.

Backup Procedures

Perform Backups Often

Backup procedures are an essential part of database administration. To help protect the integrity of your database, you should perform backups as often as possible. As a general rule, always back up the database:

- At regular, scheduled intervals, such as daily or weekly
- Before and after structural changes to the database
- Whenever you initialize journal files
- Since automatic recovery is not available in local mode, before and after executing an application run in local mode.

Design a Backup Plan

To ensure that your backup procedures meet the data processing needs of your company, you need to decide how often to take backups and how long to retain them. Develop a schedule and procedures for performing backups and stick to it.

General Guidelines

The following list identifies some guidelines to follow in designing a backup plan:

- Define the backup and recovery requirements for an application while the application is being designed. Test all backup and recovery procedures before the application is put into production.
- Make sure you backup the database after making changes to its physical definition (such as changing the page size, page range, and so on).
- Identify all archive files created since the last backup.
- If you need to concatenate archive tapes for historical records, make sure that the tapes included in the concatenation are not required for recovering the database. For example, you might concatenate the archive tapes from the previous week at the end of the current week.
- Bear in mind that restoring a database from a date several weeks in the past can be a very time-consuming process because of the volume of journal data that needs to be processed.

BACKUP Utility Statement

The examples outlined in this chapter use the BACKUP utility statement provided with CA IDMS/DB to backup the database. You can use other utilities (such as IEBGENER in z/OS) to perform the backup and recovery operation provided they restore disk files to the state they were in when copied.

If you use the CA-provided BACKUP utility statement for regularly scheduled backups, specify the FILE option rather than the AREA option. FILE lets you recover an individual file in the event it is damaged rather than having to recover the entire area. Use the AREA option only if multiple areas are stored in a single file.

Back Up After a Normal System Shutdown

Steps

While the system is inactive, back up the database using the following procedure:

Action	Statement
Offload all the non-empty journal files	ARCHIVE JOURNAL utility statement with the ALL or AUTOALL option
Copy all files associated with the database	BACKUP utility statement or any comparable backup utility

Backup While the DC/UCF System is Active

Types of Backup While System is Active

There are two types of backup that can be done while DC/UCF remains active:

- A quiesced backup during which no updates are made to the areas being copied
- A hot backup during which the areas that are copied are updated by transactions executing within the central version

While it is preferable to back up a database when it is quiesced, a site with high-availability requirements may not be able to disable updates long enough to complete the backup.

Considerations

If you decide to use a hot backup strategy, consider the following:

- The time to recover using a hot backup may be longer than with a backup produced while the area was quiesced due to additional steps in the recovery process.
- To recover using a file produced during a hot backup, all archive journal files created while the backup was taking place must also be available; without these files, the backup file cannot be used. Although the EXTRACT JOURNAL utility statement can be used to preprocess the journal images generated during this time period, the original archive files must also be available to perform a successful recovery.
- To ensure the availability of the archive journal files you should treat them in the same way as the backup file; for example, if a copy of the backup file is sent offsite, a copy of all corresponding archive files should also be sent offsite.

Note: For more information about the impact of a hot backup on recovering a database, see 21.5, “Manual Recovery".

Quiesced Backup Procedure

The procedure outlined below describes how to perform a quiesced backup.

Action	Steps
Quiesce update activity in the target areas. (See considerations)	Issue one or more of the following commands: ■ DCMT VARY AREA ... RETRIEVAL ■ DCMT VARY AREA ... OFFLINE ■ DCMT QUIESCE AREA ... ■ DCMT VARY SEGMENT ... RETRIEVAL ■ DCMT VARY SEGMENT ... OFFLINE ■ DCMT QUIESCE SEGMENT ... ■ DCMT QUIESCE DBNAME ... ■ DCMT VARY RUN UNIT ... OFFLINE
Note the quiesce point	Record the date and time that the areas were quiesced. Optionally force a new archive journal file to be created: ■ Issue a DCMT VARY JOURNAL command ■ Execute the ARCHIVE JOURNAL utility statement

Action	Steps
Copy all files containing the target areas.	Execute the BACKUP utility statement using the FILE option or any comparable backup utility.
Restart update activity in the target areas.	Issue one or more of the following commands: ■ DCMT VARY AREA ... ONLINE ■ DCMT VARY SEGMENT ... ONLINE ■ DCMT VARY ID ... TERMINATE ■ DCMT VARY RUN UNIT ... ONLINE

Hot Backup Procedure

The procedure for a hot backup is similar to that for a quiesced backup, except that updates are re-enabled before the backup is complete. The procedure described next includes establishing a second quiesce point. This is not necessary if the appropriate recovery procedure is followed.

Note: For more information about the impact of a hot backup and a second quiesce point on recovery, see 21.5, "Manual Recovery".

Action	Steps
Quiesce update activity in the target areas. (See considerations)	Issue one or more of the following commands: ■ DCMT VARY AREA ... RETRIEVAL ■ DCMT VARY AREA ... OFFLINE ■ DCMT QUIESCE AREA ... ■ DCMT VARY SEGMENT ... RETRIEVAL ■ DCMT VARY SEGMENT ... OFFLINE ■ DCMT QUIESCE SEGMENT ... ■ DCMT QUIESCE DBNAME ... ■ DCMT VARY RUN UNIT ... OFFLINE
Note the quiesce point	Record the date and time that the areas were quiesced. Optionally force a new archive journal file to be created: ■ Issue a DCMT VARY JOURNAL command ■ Execute the ARCHIVE JOURNAL utility statement

Action	Steps
Restart update activity in the target areas.	Issue one or more of the following commands: ■ DCMT VARY AREA ... ONLINE ■ DCMT VARY SEGMENT ... ONLINE ■ DCMT VARY ID ... TERMINATE ■ DCMT VARY RUN UNIT ... ONLINE
Copy all files containing the target areas.	Execute the BACKUP utility statement using the FILE option or any comparable backup utility.
Optionally, establish a second quiesce point for the target areas.	Issue one or more of the following commands: ■ DCMT VARY AREA ... RETRIEVAL ■ DCMT VARY AREA ... OFFLINE ■ DCMT QUIESCE AREA ... ■ DCMT VARY SEGMENT ... RETRIEVAL ■ DCMT VARY SEGMENT ... OFFLINE ■ DCMT QUIESCE SEGMENT ... ■ DCMT QUIESCE DBNAME ... ■ DCMT VARY RUN UNIT ... OFFLINE
Mark the end of the backup process.	Force a new archive journal file to be created: ■ Issue a DCMT VARY JOURNAL command ■ Execute the ARCHIVE JOURNAL utility statement If a second quiesce point was established, record its date and time.
If a second quiesce point was established, restart update activity in the target areas.	Issue one or more of the following commands: ■ DCMT VARY AREA ... ONLINE ■ DCMT VARY SEGMENT ... ONLINE ■ DCMT VARY ID ... TERMINATE ■ DCMT VARY RUN UNIT ... ONLINE

Quiescing Update Activity

Both DCMT VARY AREA (and SEGMENT) and DCMT QUIESCE can be used to quiesce update activity in one or more areas of the database. Consider the following when choosing which of these to use:

- If DCMT VARY is used, tasks which subsequently attempt to access a target area in an update mode (or any mode if the area is varied offline) will receive an 0966 error status. Unless the application program handles this condition, the associated task will fail. If DCMT QUIESCE is used, such tasks will wait until update activity is restarted, unless their quiesce wait time is exceeded.
- DCMT QUIESCE provides more control over the quiesce operation. For example, it is possible to specify how long the quiesce operation should wait for conflicting tasks to finish and what action should be taken in the event that the quiesce point has not been reached in the specified time interval.
- In a data sharing environment, DCMT QUIESCE will quiesce update activity across all members of the data sharing group. DCMT VARY will quiesce update activity only within the DC/UCF system in which it is executed.
- DCMT QUIESCE can be used to automate much of the backup process.

More Information

- For more information about backup automation, see 21.2.4, "Automating the Backup Process".
- For more information about the DCMT system task, see the *CA IDMS System Tasks and Operator Commands Guide*.

Quiescing Update Activity for System Areas

When backing up a system area, such as a load area, it may be necessary to terminate predefined system run units by issuing a DCMT VARY RUN UNIT ... OFFLINE command. This will be necessary if predefined run units for the target area have been defined in the system definition and such run units access the area in update mode. You can determine this by issuing a DCMT DISPLAY RUN UNIT command.

Varying a system run unit offline does not prevent overflow run units from being started to service requests for the area. It simply terminates predefined run units of the specified type. Since varying an area offline will impact the system's ability to service requests for the area, it is advisable to quiesce update activity to system areas either by varying their status to retrieval or by using the DCMT QUIESCE command.

Depending on the options specified when issuing a DCMT VARY AREA, DCMT VARY SEGMENT, or DCMT QUIESCE command, the system may automatically terminate conflicting predefined system run units.

Note: For more information about when predefined system run units are automatically terminated, see the individual commands in the *CA IDMS System Tasks and Operator Commands Guide*.

Data Sharing Considerations

In a data sharing environment, whenever update activity is quiesced, it must be quiesced in all DC/UCF systems that are members of the data sharing group. If a DCMT QUIESCE command is used, then update activity will automatically be quiesced on all members within the group. If a DCMT VARY AREA or DCMT VARY SEGMENT command is used, it must be executed on each system that is a member of the group. This can be accomplished by broadcasting the DCMT command.

Note: For more information about broadcasting DCMT commands, see the *CA IDMS System Tasks and Operator Commands Guide*.

Back Up Before and After Local Mode Jobs

Two Options

To protect data to be accessed by an update job running in local mode, you can either:

- Use local mode journaling. This option is best for large databases that would require a long time to backup and restore.
- Back up the database before and after you run the job. This option is best for small databases that can be backed up within a reasonable time frame.

Note: For information about local mode journaling, see Chapter 19, “Journaling Procedures”.

Steps to Back Up the Database

Follow the steps below to back up a database before and after running an update application in local mode:

Action	Steps
Make the areas to be accessed by the application unavailable under the central version	DCMT VARY AREA or SEGMENT with the OFFLINE, RETRIEVAL, or TRANSIENT RETRIEVAL option
Before running an application, back up each file of the database	BACKUP or any comparable backup utility
Dummy the journal file by adding a dummy file definition statement in the execution JCL of the local mode application if the DMCL being used has a tape journal file defined	
After running the application,back up each file of the database	BACKUP or any comparable backup utility
Swap to another disk journal file to coordinate CVs archive journal files with the backup	DCMT VARY JOURNAL
Re-activate the areas for use under the central version	■ If the areas are OFFLINE or in RETRIEVAL mode,issue DCMT VARY AREA or SEGMENT ONLINE ■ If the areas are in TRANSIENT RETRIEVAL mode, first vary them OFFLINE and then ONLINE.

CA ADS: When you vary an area in preparation for a local mode update, CA ADS users should vary the area to either OFFLINE or TRANSIENT RETRIEVAL mode; do *not* use RETRIEVAL mode.

Automating the Backup Process

Exploiting DCMT QUIESCE

Backing up a database while the DC/UCF system is active can be automated through the use of the DCMT QUIESCE command. To assist in this effort, the following can be specified as options:

- A unique identifier for use in subsequent DCMT DISPLAY ID and DCMT VARY ID commands to query or terminate an outstanding quiesce operation.
- The action that should be taken in the event that a quiesce point cannot be reached within a specified time interval. The available choices are to abandon the quiesce operation or force the quiesce by canceling conflicting tasks.
- An indication of whether a new archive journal file should be created when the quiesce point is reached.
- An indication of whether update activity in the target areas should be restarted automatically once the areas are quiesced.

Quiesce User Exit

When a quiesce point is achieved, numbered exit, Exit 38 is invoked. This exit can be used to initiate the next step in the backup process. For example, it can submit a job to the internal reader, thus enabling the QUIESCE task to automatically initiate a copy operation. Once the files are copied, a subsequent UCF batch job step can invoke further system tasks to complete the backup process.

Rather than submitting a batch job, exit 38 might instead use an API to directly interface to a "zero-time copy" facility if the database resides on a storage device that provides such a capability.

More Information

- For more information about how to code an Exit 38 routine, see the *CA IDMS System Operations Guide*.
- For more information about the DCMT QUIESCE command, see the *CA IDMS System Tasks and Operator Commands Guide*.

Automating a Quiesced Backup

The following illustrates how the DCMT QUIESCE command can be used to automate a quiesced backup operation.

Activity	Description
dcmt quiesce dbname CUST hold swap CUSTBKP	This command initiates a quiesce operation identified as CUSTBKP. All areas in all segments included in the database name CUST will be quiesced. When the quiesce point is reached, a new archive journal file will be created and exit 38 will be invoked. The quiesce point will be held until the quiesce operation is explicitly terminated.
Exit 38 is invoked	Exit 38 submits a batch job through the internal reader (or an equivalent mechanism) to initiate the copy operation.
Batch job is executed	The batch job first copies all files containing areas of the CUST database and then invokes a UCF batch job step that terminates the quiesce operation by issuing a DCMT VARY ID command.
dcmt vary id CUSTBKP terminate	This command terminates the quiesce operation and makes the CUST areas available for update.

Automating a Hot Backup

The following illustrates how the DCMT QUIESCE command can be used to automate a hot backup operation.

Activity	Description
dcmt quiesce dbname CUST nohold swap CUSTBKP1	This command initiates a quiesce operation identified as CUSTBKP1. All areas in all segments included in the database name CUST will be quiesced. When the quiesce point is reached, a new archive journal file will be created and exit 38 will be invoked. The quiesce operation will then terminate and make the areas available for update.
Exit 38 is invoked	Exit 38 submits a batch job through the internal reader (or an equivalent facility depending on the operating system) to initiate the copy operation.

Activity	Description
Batch job is executed	The batch job first copies all files containing areas of the CUST database and then invokes a UCF batch job step. The UCF batch job step either initiates a second quiesce operation by issuing a DCMT QUIESCE command or forces a new archive journal file to be created by issuing a DCMT VARY JOURNAL command.
dcmt quiesce dbname CUST nohold swap CUSTBKP2	This command initiates a quiesce operation identified as CUSTBKP2. All areas in all segments included in the database name CUST will be quiesced. When the quiesce point is reached, a new archive journal file will be created and exit 38 will be invoked. The quiesce operation will then terminate and make the areas available for update. Exit 38 examines the quiesce identifier and determines that no further action is needed.
dcmt vary journal	This command forces the use of another disk journal file which in turn causes a batch execution of the ARCHIVE JOURNAL utility statement. **Note:** Automatic submission of the ARCHIVE JOURNAL job is dependent on the implementation of a site-specific means (such as WTOEXIT) to examine console messages and use operating system facilities to submit a batch job.

Automatic Recovery

Available Only Under the Central Version

Automatic recovery is available only under the central version. Automatic recovery occurs when CA IDMS/DB:

- *Warmstarts,* following a system failure
- *Automatically rolls back* a failing transaction
- *Automatically rolls back* the changes made by a physical DDL or SQL statement that encountered errors

Warmstart

Due to System Failure

Warmstart occurs when CA IDMS starts up and, by examining the journal files, it detects that the previous execution of the DC/UCF terminated abnormally CA IDMS uses the journal files to rollback or restart all transactions that were active when the system failed.

How You Respond to a System Failure

In response to a DC/UCF system failure, you should immediately restart the system. In a data sharing environment, or if distributed transactions were active at the time of failure, it is particularly important to restart failing systems as soon as possible, since some data may be inaccessible within other systems until the failing system has completed its warmstart.

Note: Do *not* offload any journal files between the time of system failure and your first attempt to warmstart the system. If you must offload, use the READ option of the ARCHIVE JOURNAL utility statement.

Data Sharing Considerations

In general, you respond to a DC/UCF system failure in the same way regardless of whether the system is a member of a data sharing group. However, certain types of failures, such as a loss in connectivity to a coupling facility, require special action. Additionally, if a member is unable to warmstart and manual recovery becomes necessary, then data sharing introduces additional considerations.

More Information

- For more information about recovery considerations in a data sharing environment, see the *CA IDMS System Operations Guide*.
- For more information about the impact of data sharing to manual recovery, see 21.5, “Manual Recovery”.

Incomplete Distributed Transactions at Startup

When restarting a failed central version, warmstart identifies incomplete distributed transactions that were active at the time of failure. Depending on where in the commit process the failure occurred, these transactions are completed by warmstart or are restarted. If restarted, the transactions remain active until resynchronization takes place with the other resource or transaction managers involved in the transaction or until the transactions are manually completed.

If a restarted transaction is in an InDoubt state, then any locks held by that transaction at the time of failure are reacquired and held until the transaction is completed. Since these locks prevent access to resources that were updated by the transaction, it is important to restart all failed systems as soon as possible in order that resynchronization can complete the transaction and free the locks.

Note: For more information about recovering distributed transactions, see 21.3.3, “Resynchronization” and 21.4, “Distributed Transaction Recovery Considerations”.

The following sample messages might be displayed when a distributed transaction is restarted:

```
IDMS DC202038 V74 In-Doubt Transaction-ID 1416 will be added to the
unrecovered transaction list
IDMS DC202051 V74 Warmstart COMPLETE, but recovery of SOME transactions
have been DEFERRED until later in the startup
IDMS DB342017 V74 T1 Will lock Transaction-ID 1416
IDMS DB342019 V74 T1 DTRID SYSTEM74::01650C90A708A9B2-01650C8C4207D9FF
active at startup
IDMS DB342020 V74 T1 DTRID SYSTEM74::01650C90A708A9B2-01650C8C4207D9FF
has been restarted
IDMS DB342022 V74 T1 In-Doubt Transaction 1416 has been restarted
```

Incomplete Warmstart

Certain errors, such as I/O errors or open failures, may prevent warmstart from rolling out the changes in one or more database files. If this occurs, warmstart will continue, the system will start up and the transactions affected by the error will be restarted. Once restarted, automatic rollback will be invoked to again attempt to remove the effect of the unrecovered transactions. If automatic rollback is successful, no further action is necessary although the reason for the original failure should be investigated and corrective action taken if necessary. If automatic rollback is not successful, the unrecovered transactions will be suspended just as if they had encountered an I/O error. To correct the situation, You respond as if a database file I/O error occurred. First take whatever action is necessary to make the file available, such as restoring a damaged file or using DCMT commands to correct a data set name. Then restart the suspended transactions by issuing a DCMT VARY FILE ACTIVE command.

Note: For more information about responding to I/O errors, see 21.7, "Recovery Procedures from Database File I/O Errors".

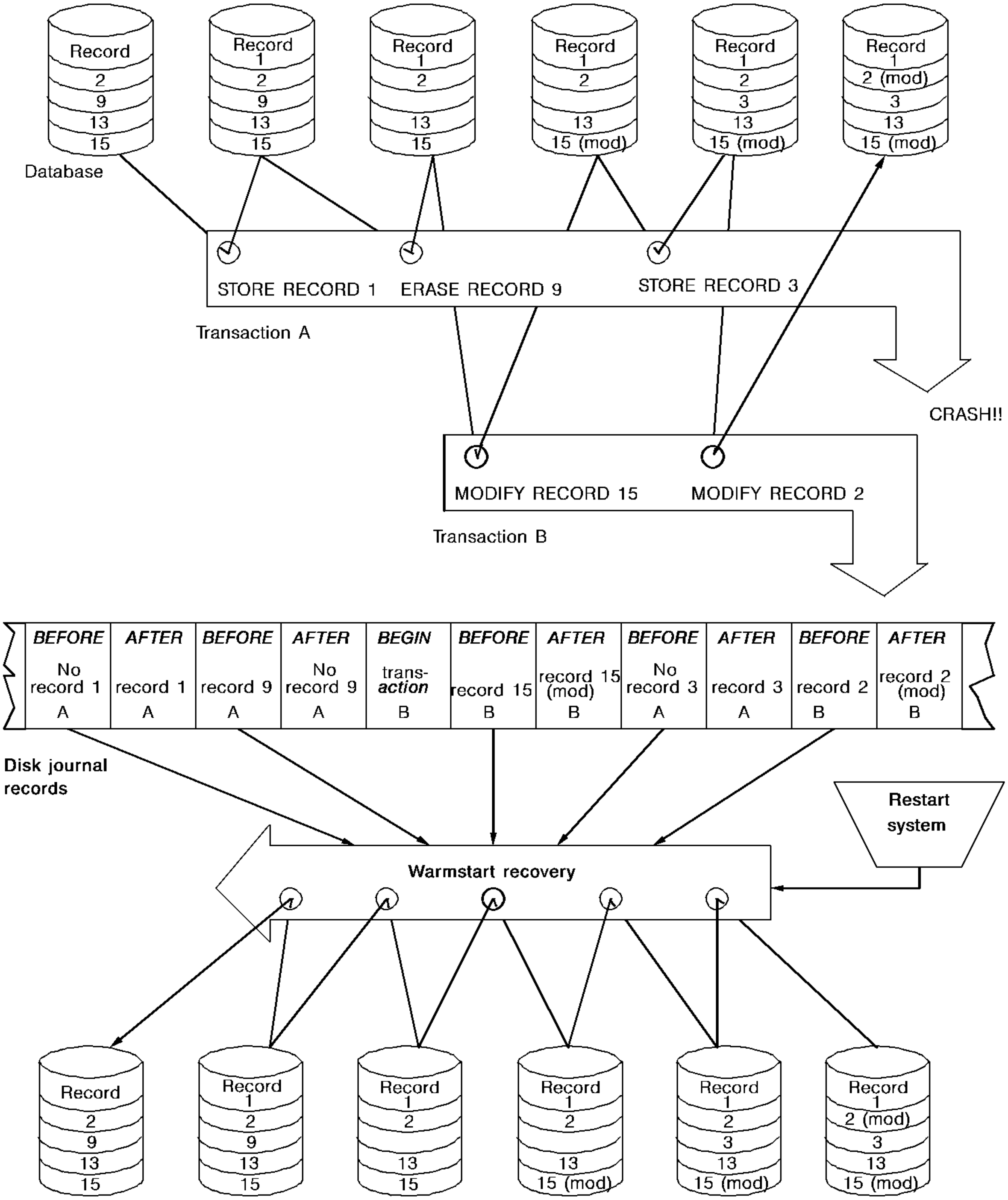
How Warmstart Works

During warmstart, CA IDMS/DB does the following:

1. Establishes which disk journal file was active at the time of the failure
2. Locates the last journal record written before the system failed
3. Either restarts or rolls back and writes ABRT checkpoints for all incomplete transactions.

Example

The following example shows how a warmstart operation is done. In this example, the two transactions are active at the time of the system crash. Both are recovered automatically when the system is restarted.



Automatic Rollback

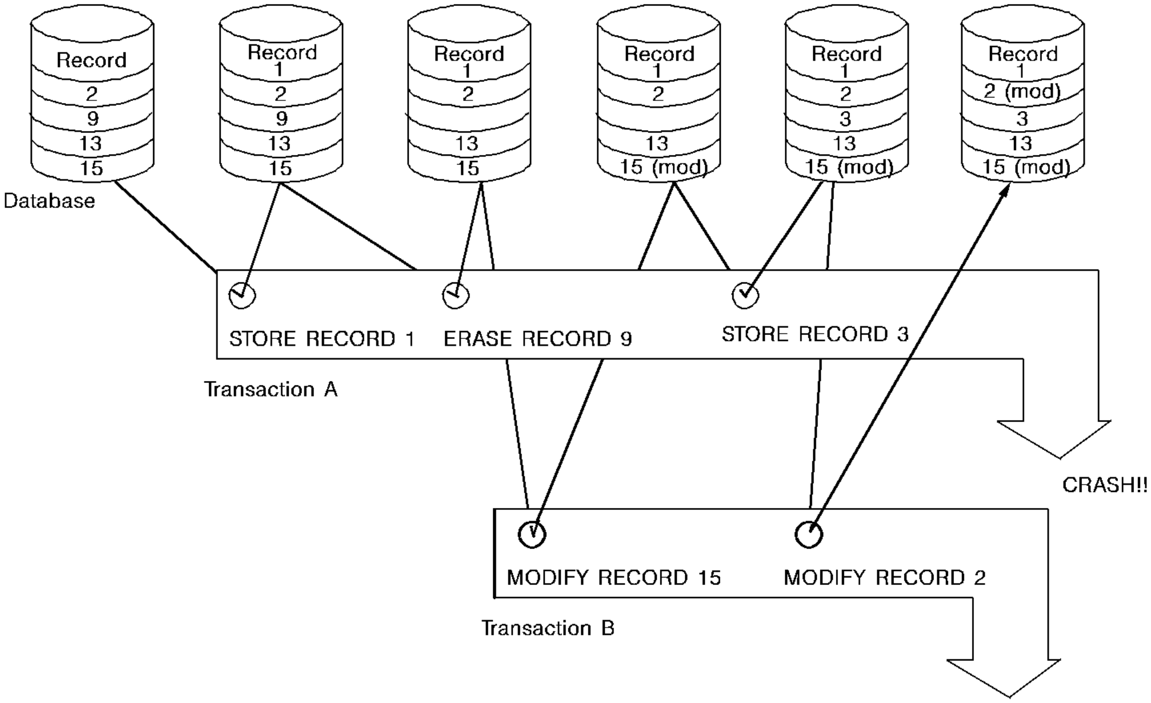
Due to Transaction Failure

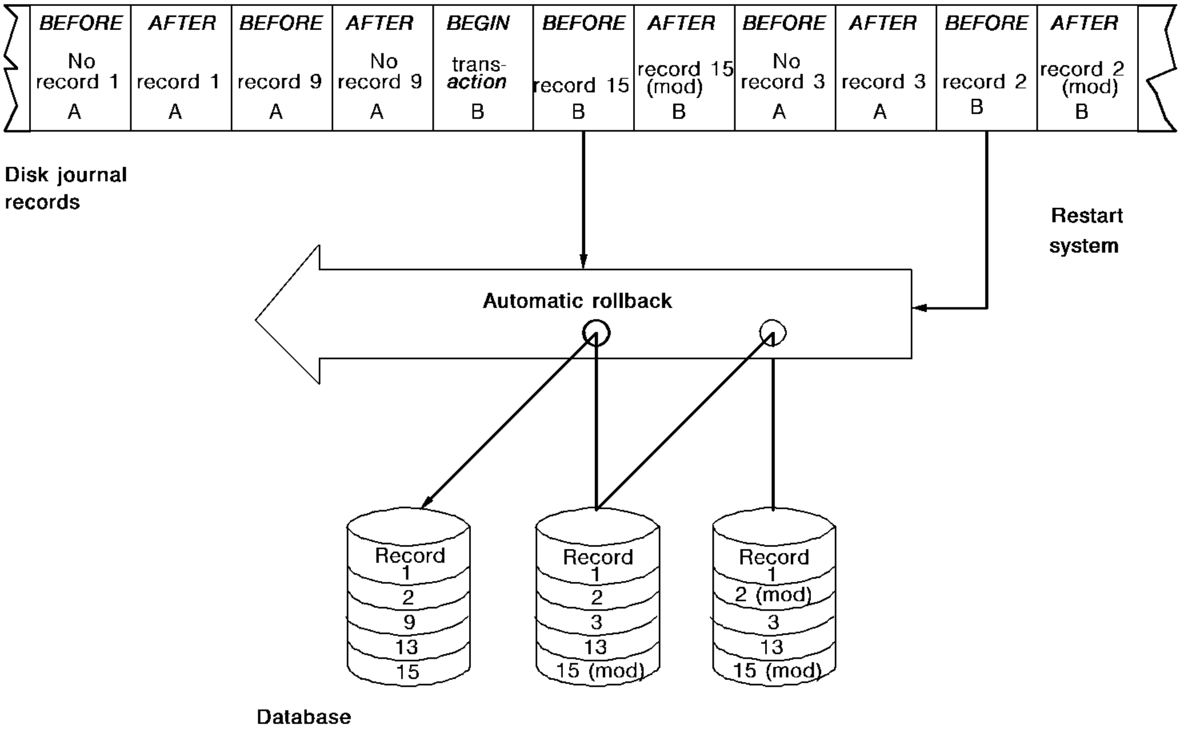
Automatic rollback occurs when a transaction fails or an application requests recovery by means of the ROLLBACK command. CA IDMS/DB writes an ABRT checkpoint for the transaction and automatically rolls out the changes made to the database by the transaction. The recovery occurs while the system continues to process requests by other concurrently active transactions.

Note: Automatic rollback also occurs when an error is encountered executing SQL or physical DDL statements. In these cases, an RTSV checkpoint record is written instead of an ABRT checkpoint, but in other respects, the two recovery operations are the same.

Example

The following example shows how an automatic rollback occurs. In this example, transaction B aborts. CA IDMS/DB then performs an automatic rollback for transaction B while other transactions continue to process.




BEFORE
No record 1
A
AFTER
record 1
A
BEFORE
record 9
A
AFTER
No record 9
A
BEGIN
trans-action
B
BEFORE
record 15
B
AFTER
record 15 (mod)
B
BEFORE
No record 3
A
AFTER
record 3
A
BEFORE
record 2
B
AFTER
record 2 (mod)
B
Disk journal records
Restart system
Automatic rollback
Record 1
2
9
13
15
Record 1
2
3
13
15 (mod)
Record 1
2 (mod)
3
13
15 (mod)
Database

Resynchronization

What is Resynchronization?

Resynchronization is a process in which information is exchanged between two systems to establish attributes relevant to the two-phase commit process and to complete outstanding distributed transactions following a failure. This chapter focuses on resynchronization between CA IDMS systems.

Note: For information about resynchronization between CICS or RRS and CA IDMS see the *CA IDMS System Operations Guide*.

When Does It Occur?

Resynchronization between CA IDMS systems occurs at the following times.

- When a central version is started, resynchronization is initiated with each known backend system. A backend system is known if it was accessed by a database session since the last time the journal files were formatted. If the started system cannot communicate with one or more of its backend systems, resynchronization with those systems is retried on a periodic basis until communication is reestablished.
- When a remote database session is started, resynchronization is initiated with the backend system if it was previously unknown (that is, if this is the first time the backend system has been accessed since the journal files were formatted) or if the backend system has been recycled since resynchronization previously took place between the two systems.

  Note: A remote database session is started when an application binds a run unit or connects an SQL session to a remote database. It is also started when a DCUF task establishes a remote dictionary as a default.
- When manually driven through a DCMT VARY DISTRIBUTED RESOURCE MANAGER command, resynchronization is attempted with the specified resource manager.

Note: For more information about DCMT and DCUF, see the *CA IDMS System Tasks and Operator Commands Guide*.

What Does It Entail?

Resynchronization begins with an exchange of startup times and journal timestamps between the two systems. As the name implies, the startup time is the time at which a system was started and is used to detect when a partner system is recycled.

The journal timestamp is assigned by a central version the first time it opens a set of journal files after they have been formatted. It is subsequently used to detect when a partner's journal files have been reformatted since the last time the two systems resynchronized with each other.

If no distributed transactions involving the two systems exist at the time that resynchronization takes place, the two systems simply exchange the above information, update their journal files with new or changed partner information, and record each other as open resource managers.

If distributed transactions involving the two systems do exist at the time of resynchronization, each system compares its partner's current journal timestamp with the one that it had saved previously. If the timestamps are the same, resynchronization proceeds by exchanging information about the incomplete distributed transactions that are pending resynchronization. If the timestamps are not the same, it indicates that one of the following has occurred:

- The partner system's journal files have been prematurely formatted.
- The partner system has been started with incorrect journal files.
- The partner system has been started with an incorrect DCNAME parameter.

Any of these conditions result in a resynchronization failure.

Responding to Resynchronization Failures

If resynchronization detects a journal stamp mismatch with a system for which incomplete distributed transactions exist, resynchronization cannot complete. When this occurs, messages are displayed that show the old and new journal stamps and the incomplete distributed transactions that are impacted by the mismatch. The operator is prompted as to what action should be taken. The following example shows the messages that are displayed as a result of a mismatch in SYSTEM74's journal stamps as they are known to SYSTEM73.

```
DC329021 V73 T23 Journal stamp mismatch for SYSTEM74::DSI_SRV *OLD
2002-12-14-07.20.36.376737
DC329021 V73 T23 Journal stamp mismatch for SYSTEM74::DSI_SRV *NEW
2003-01-30-08.07.42.278334
DC329022 V73 T23 RM Name          Dtrid                    Branch
          State
DC329023 V73 T23 SYSTEM74::DSI_SRV SYSTEM74::01650D6EDFB1AB93-
01650D6A79F31E50 InDoubt
DC329024 V73 REPLY 01 T23 Reply with resynchronization action for
SYSTEM74::DSI_SRV (Ignore,Defer):
```

Before replying to message DC329024, the cause of the mismatch should be determined. The appropriate response should then be made as outlined in the following table. Until a response is made to the DC329024 message, no database access is permitted with the identified resource manager. Any task attempting such access waits until a response has been made or its wait time is exceeded.

Reply	Meaning and Considerations
IGNORE	This reply specifies that resynchronization with the resource manager should continue. The distributed transactions listed in the preceding DC329023 messages require manual completion. IGNORE is appropriate if the partner system's journal files have been prematurely formatted. In this case, the only way to complete the affected transactions is to do so manually, since the journal entries required to complete the transactions automatically are no longer available on the partner system's journal files. For guidance on how to manually complete the transactions, see "Completing Transactions Manually".

Reply	Meaning and Considerations
DEFER	This reply specifies that resynchronization with the resource manager should be postponed until a later time. Database access with the identified resource manager is disallowed until resynchronization has completed successfully. DEFER is appropriate if the mismatch can be corrected by recycling one or the other system. Perhaps one of the systems was started with incorrect journal files or the partner system was started with an incorrect DCNAME parameter. After replying DEFER, the system in error should be shutdown and restarted correctly. It may then be necessary to initiate resynchronization using a DCMT VARY DISTRIBUTED RESOURCE MANAGER command.

Distributed Transaction Recovery Considerations

Recovery is Automatic

The primary responsibility for effecting recovery of distributed transactions lies with warmstart and resynchronization. Consequently, manual intervention should almost never be required provided that correct operating procedures are followed.

System Interdependence

The one important consideration when dealing with distributed transaction recovery is that systems are no longer independent with respect to recovery. Information on a coordinator's journal files might be needed to complete the recovery process for a participant.

Restarting Failed Systems

When restarting a failed central version, it is advisable to restart it on the same logical operating system image as the one on which it abnormally terminated. This ensures that the restarted system can access (and be accessed by) the same systems with which it was able to communicate prior to the abnormal termination regardless of the intersystem access methods available for use. If the restarted system cannot communicate with another system, it is not able to resynchronize with that system. This may leave incomplete transactions holding locks that prevent access to portions of the database. Resynchronization will eventually complete these transactions when the necessary intersystem communications are re-established.

Completing Transactions Manually In rare circumstances following a resynchronization failure, it may be necessary to complete a distributed transaction manually. CA IDMS provides two ways to do this: either by using DCMT commands or through facilities provided by the manual recovery utilities. Only transactions that are pending resynchronization should be completed manually. This restriction is enforced if using the DCMT commands.

Completing InDoubt Transactions

Regardless of how it is done, when manually completing a transaction whose state is InDoubt, you must specify whether to commit or back out the transaction's changes. You should research the situation carefully before taking any action. If you make the wrong decision, the distributed transaction will have a mixed outcome, meaning that some of its changes will be committed while others will be backed out.

The following sources of information might be helpful in determining the correct action to take:

- The output from a DCMT DISPLAY DISTRIBUTED TRANSACTION command. This will indicate what system is acting as the coordinator for the transaction and can be used if the transaction is still active within the participant. Similar information can be obtained from the detailed reports produced by the PRINT JOURNAL or FIX ARCHIVE utilities.
- Facilities provided by the coordinator for determining the outcome of a transaction.
  - If the coordinator is a CA IDMS system, its journal files will contain a DCOM record for the transaction if its changes should be committed. Use the PRINT JOURNAL summary report to see all incomplete distributed transactions or J-Report 8 to see a list of all distributed checkpoints. (Before generating either report, be sure to offload and include all journal files created since the point of failure.) If there is no DCOM entry for the transaction, then the transaction's changes should be backed out.
  - If the coordinator is RRS, use the RRS ISPF panels to determine the outcome of the transaction. For more information about RRS panels, see the IBM guide *MVS Programming: Resource Recovery*.
  - If the coordinator is CICS, examine its LOG file or use CEMT commands to determine the outcome of the transaction.

Completing Distributed Transactions Using DCMT

Completing InDoubt Transactions Using DCMT

Once you have determined whether an InDoubt transaction's changes should be committed or backed out, issue a DCMT VARY DISTRIBUTED TRANSACTION command specifying COMMIT or BACKOUT as appropriate.

Completing a transaction in this way will mark it as heuristically committed or backed out (with an outcome of HC or HR respectively). After the DCMT command is issued, the transaction will hold no locks, but it will remain active until either:

- Resynchronization with the coordinator takes place, or
- The transaction is forced to end by a DCMT VARY DISTRIBUTED TRANSACTION command that specifies FORGET.

If resynchronization is allowed to complete the transaction, a check will be made to see if its overall outcome is consistent (meaning that the transaction's changes were either all committed or all backed out). If a mixed outcome is detected, this fact will be reported on the log and the transaction will remain active on the coordi nator until a DCMT command is issued causing it to be forgotten.

Completing InCommit or InBackout Transactions Using DCMT

A transaction whose state is InCommit or InBackout holds no locks and, therefore, is not preventing access to any part of the database. Consequently, there is no urgency involved in completing such transactions and they should normally be allowed to complete automatically through resynchronization. However, if resynchronization fails due to an uncorrectable error, such as premature formatting of a journal file, the DCMT VARY DISTRIBUTED TRANSACTION command specifying FORGET can be used to force completion of InCommit or InBackout transactions.

Note: Even specifying FORGET will not cause a transaction to complete if it cannot successfully communicate with all resource managers that still require notification.

Incomplete Transactions and Manual Recovery

The Impact of Distributed Transactions

If manual recovery becomes necessary, the process is generally the same regardless of whether the archive journal files contain distributed transaction checkpoint records or not.

However, special action may be needed when a ROLLFORWARD operation terminates or a ROLLBACK operation begins at a point in time when a distributed transaction is active and in an InDoubt state. The problem that arises in these situations is that the recovery utility does not know whether to commit the local changes made by the InDoubt transaction or back them out. Since the utility has no way of communicating with a coordinator to determine what action to take, it may be necessary for the DBA to explicitly specify the final outcome for the transaction.

Determining If InDoubt Transactions Exist

All recovery utilities that report transaction information, also report on distributed transactions. For example, the PRINT JOURNAL and FIX ARCHIVE utilities include information about all distributed transactions in their detailed report and list incomplete transactions in their summary report. Incomplete transactions that are in an InDoubt state will be among those listed in the summary report.

Why There are Incomplete InDoubt Transactions

A distributed transaction is in an InDoubt state when the last journal record written for that transaction is a DIND. Normally, a DCOM or a DBAK record follows a DIND, and its presence determines whether a transaction's changes should be committed or back out respectively. The absence of a DCOM or DBAK record may be because:

- It exists but on a later archive journal file that is not being processed in the current execution of the recovery utility.
- It exists but is split between two archive journal files, only the first of which is being processed in the current execution of the recovery utility.
- It has not been written because resynchronization with the transaction's coordinator has not completed.

The first two conditions may indicate that not all required journal records are being processed. The third condition may necessitate an explicit specification of how to complete the transaction.

Note: For more information about the journal records that are written in support of distributed transactions, see "Two-Phase Commit Journaling" in 19.2.4, “Two-Phase Commit Journaling”.

How Utilities Deal with InDoubt Transactions

By default, the recovery utilities leave an InDoubt transaction in its InDoubt state, meaning that its changes are not rolled out. A DBA can override this default behavior by adding an entry to a manual recovery control file to explicitly specify the action to be take for an InDoubt transaction.

When to Manually Complete Transactions

Explicitly overriding the default action should normally not be necessary. In fact, the presence of InDoubt transactions at the end of a ROLLFORWARD or the beginning of a ROLLBACK operation should be researched to determine the reason for their existence and to ensure that the recovery procedure being followed is valid and includes all necessary journal input.

For example, an InDoubt transaction might validly be encountered when recovering a damaged database file. In this case, the transaction should be allowed to remain InDoubt. When the recovered file is subsequently varied active to the central version, the transaction will be completed (backed out or committed) automatically.

The only time that an InDoubt transaction should be explicitly completed is in exceptional situations that prevent resynchronization from completing the transaction automatically, such as:

- When a coordinator is permanently inaccessible
- When a coordinator's journal files have been prematurely formatted
- When a participant's journal files have been damaged or prematurely formatted.

Even in the first two situations, if the transaction is still active within the participant central version it should be completed using a DCMT VARY DISTRIBUTED TRANSACTION command rather than using a manual recovery control file override.

How to Explicitly Complete InDoubt Transactions

Before taking any action to complete an InDoubt transaction, you must first determine whether its changes should be committed of backed out. This will typically require using facilities provided by the coordinator to determine the final transaction outcome.

Note: See "Distributed Transaction Recovery Considerations" in 21.4, “Distributed Transaction Recovery Considerations” for potential sources of information.

To explicitly complete an InDoubt transaction, an entry must be added to a manual recovery control file. The following utilities will read a manual recovery control file for this purpose:

- EXTRACT JOURNAL (unless ALL is specified)
- FIX ARCHIVE
- MERGE ARCHIVE (if COMPLETE is specified)
- PRINT JOURNAL
- ROLLBACK
- ROLLFORWARD (unless ALL is specified)

To complete a transaction, you must specify its DTRID and an action of either COMMIT or BACKOUT.

If an InDoubt transaction is completed by a utility that creates an output archive file (FIX ARCHIVE, EXTRACT JOURNAL and MERGE ARCHIVE), the utility will write additional distributed transaction checkpoint records that complete the transaction in the specified way.

Note: For more information about the format and use of a manual recovery control file, see the above utilities in the *CA IDMS Utilities Guide*.

Deleting Resource Managers

When to Delete a Resource Manager

A resource manager should only be deleted if it no longer exists or if it is permanently inaccessible. Even in these cases, there is often no need to explicitly delete a resource manager, since it will disappear when the journal files are next formatted. However, if incomplete, distributed transactions exist that involve an inaccessible resource manager as a participant, then you may want to explicitly delete the resource manager to enable the transactions to be completed.

How to Delete Resource Managers

You can delete a resource manager by issuing a DCMT VARY DISTRIBUTED RESOURCE MANAGER command specifying DELETE. This removes the resource manager from the system, purges it from the journal files and deletes all associated transaction interests. Clearly, this command should be used with care.

The following procedure should be followed to delete a resource manager:

1. Obtain a list of transactions in which the resource manager has an interest by issuing a DCMT DISPLAY DISTRIBUTED RESOURCE MANAGER command for the target resource manager.
2. For each listed transaction determine whether the resource manager is a coordinator or a participant by displaying its detail using a DCMT DISPLAY DISTRIBUTED TRANSACTION command.
3. Complete each transaction for which the resource manager is a coordinator by issuing one or more DCMT VARY DISTRIBUTED TRANSACTION commands.
4. Issue a DCMT VARY RESOURCE MANAGER ... DELETE to delete the resource manager.
5. Complete each transaction for which the resource manager was a participant by issuing a DCMT VARY DISTRIBUTED TRANSACTION command.

Note: For information about using DCMT commands to complete transactions, see 21.4.1, “Completing Distributed Transactions Using DCMT” Using DCMT.

Manual Recovery

Before You Begin

Before you attempt to manually recover the areas or files of the database, gather the available facts, such as:

1. The time of the system or transaction failure
2. Whether the failure occurred under the central version or in local mode
3. What applications were running at the time the system failed
4. Which areas of the database were in use and whether these were in update mode
5. The time of the preceding quiesce point

You can use the PRINT JOURNAL or MERGE ARCHIVE utility statements to determine the information in items 3, 4, and 5.

Locate Backup and Archive Files

After you've determined the nature of the failure, locate the most recent backup of the database and all archive journal files created since the backup.

Note: To successfully recover the database, all of the archive files must be readable. To increase the likelihood of this, you can define multiple archive files in the DMCL used to execute the ARCHIVE JOURNAL utility statement. This directs CA IDMS/DB to create multiple archive files during offload.

Determine if InDoubt Transactions Need Special Attention

Depending on the nature of the recovery operation, incomplete InDoubt transactions may need to be completed manually as part of the recovery process.

Note: For more information about dealing with InDoubt transactions during manual recovery process, see 21.4.2, “Incomplete Transactions and Manual Recovery”.

Minimize Scope of Recovery

You can limit the recovery process by recovering only the areas or files that were impacted by the failure. Areas that were available for retrieval do not have to be recovered. Depending on the nature of the failure, recovery may be restricted to an individual file. If the recovery is due to an application error, all areas updated by the application may need to be recovered to insure the logical integrity of the database. This may in turn necessitate the recovery of other areas, if another application has updated both the original and additional areas.

After You Are Done

After you recover an area or file, check the validity of the recovery by:

- Following procedures you designed to check the validity of the data; for example, by executing a report you run regularly and comparing the output to output produced before the recovery
- Verifying the structure of the database by executing the IDMSDBAN utility

Note: For more information on IDMSDBAN, see the *CA IDMS Utilities Guide*.

The remainder of this chapter describes manual recovery procedures under the following circumstances:

- After a warmstart fails
- I/O errors in a database file
- I/O errors in a journal file
- When journaling in local mode
- When using the database in both local mode and under the central version (mixed-mode recovery)

It also provides special considerations for data sharing environments and native VSAM files.

Recovery From a Quiesced Backup

Quiesced Backup

A quiesced backup is a backup that is performed while no updates are being made to the data that is being copied. The following types of backup are quiesced backups:

- A backup performed after the DC/UCF system is shutdown
- A backup performed while the DC/UCF system is active, provided that the affected areas are quiesced at the time of the backup
- A backup performed before and after a local mode job

Note: For more information about how to backup a database, see see 21.2, “Backup Procedures”.

Recovery Procedure

The procedure outlined below describes the general approach to recovery from a quiesced backup. See the later sections in this chapter for additional considerations specific to certain types of failures.

Action	Steps
Copy the files that need to be recovered from the backup **When required:** Always.	Execute the RESTORE utility statement using the FILE option or another comparable utility.
Consolidate, in the sequence in which they were created, the archive journal files created since the quiesce point established at the start of the backup procedure. **When required:** This step is necessary only under the following conditions: ■ In z/OS environments, if the subsequent ROLLFORWARD utility statement will be executed with the SEQUENTIAL option and more than one archive journal file must be processed. In a data sharing environment, if more than one member has updated the affected areas and the subsequent ROLLFORWARD utility statement will be executed with either the SEQUENTIAL or the ALL and STOP TIME options.	Execute one of the following and use as input the properly concatenated set of archive files: ■ FIX ARCHIVE utility statement ■ MERGE ARCHIVE utility statement ■ EXTRACT JOURNAL utility statement ■ another comparable utility If consolidating archive files from multiple members and the subsequent rollforward will be executed with either the SEQUENTIAL or the ALL and STOP TIME options, use the MERGE ARCHIVE utility statement. **Note:** For more information, see 21.11, “Data Sharing Recovery Considerations”. If recovery involves local mode journal files, the MERGE ARCHIVE utility statement can be used to consolidate both local mode journal files and archive files. **Note:** For more information, see 21.10, “Recovery Procedures for Mixed-Mode Operations”.

Action	Steps
Reapply to the restored files all updates made since the backup was taken **When required:** Always.	Execute the ROLLFORWARD utility statement using either the consolidated journal file or individual archive files concatenated in the sequence in which they were created. If the journal files were consolidated using the EXTRACT JOURNAL utility, specify the FROM EXTRACT option. If FROM EXTRACT is not specified, then the following considerations apply: ■ Specify the SORTED option unless there is insufficient disk space available. SORTED must be specified if: ■ A consolidated journal file is not used as input in z/OS environments and more than one archive file must be processed. ■ The input journal file is on a device, such as a disk or a 3490 that does not support reading backwards. ■ Running ROLLFORWARD in a z/VM environment. If the SEQUENTIAL option is used and the quiesce point for the affected areas does not coincide with the start of the first input file, use the START TIME parameter to identify the quiesce point.

Recovery From a Hot Backup

Hot Backup

A hot backup is a backup that is performed while the database is being updated. The steps that must be taken to create a usable hot backup are described in 21.2, “Backup Procedures”.

Recovery Procedures

Following are two approaches to recovery from a hot backup. The first involves the use of both the ROLLBACK and ROLLFORWARD utility statements; the second involves two executions of the ROLLFORWARD utility statement. Either approach can be used to successfully recover from a hot backup; however certain conditions must be satisfied to use the second approach.

Note: For additional considerations associated with specific types of failure, refer to later sections in this chapter.

InDoubt Transaction Considerations

With either approach, there is no need to take any special action with regard to incomplete InDoubt transactions during the first recovery operation (during a ROLLBACK in approach 1 or the first ROLLFORWARD in approach 2), since the utility will handle them correctly.

However, depending on the nature of the recovery, you may need to take some action for InDoubt transactions that remain at the end of the final ROLLFORWARD operation. For more information, see 21.4.2, “Incomplete Transactions and Manual Recovery”.

Restore Procedure 1

This approach can always be used to recover from a hot backup provided that the correct procedures were followed when the backup was taken and the necessary journal and backup files are available.

Action	Steps
Copy the files that need to be recovered from the backup. **When required:** Always.	Execute the RESTORE utility statement using the FILE option or another comparable utility.

Action	Steps
Identify: ■ The quiesce point that was taken at the beginning of the backup procedure. ■ The archive journal files created since this quiesce point up to and including the one created at the end of the backup procedure. ■ All archive journal files created since the quiesce point up to the point of failure. **When required:** Always.	Use the PRINT JOURNAL utility statement, or if the quiesce point was established using the DCMT QUIESCE command, examine the operating system log for the DC/UCF system on which the DCMT command was issued.
Consolidate, in the sequence in which they were created, the archive journal files created between the quiesce point and the end of the backup procedure. **When required:** This step is necessary only under the following conditions: ■ In z/OS environments if more than one input journal file must be processed. In a data sharing environment, if the SEQUENTIAL option will be specified on the subsequent ROLLBACK utility statement and more than one member's journal images must be processed.	Execute one of the following and use as input the properly concatenated set of archive files: ■ FIX ARCHIVE utility statement ■ MERGE ARCHIVE utility statement ■ Another comparable utility If consolidating archive files from multiple members and the subsequent rollback will be executed with the SEQUENTIAL option, use the MERGE ARCHIVE utility statement. **Note:** For more information, see Data Sharing Recovery Considerations (see page 666). This and the subsequent step can be combined by using a sort utility to do the consolidation unless the use of MERGE ARCHIVE is required.
If backward read is not supported, presort the journal blocks created between the quiesce point and the end of the backup procedure in reverse sequence. Multiple archive files may be consolidated into a single sorted output file. **When required:** This step is necessary in a z/VM environment or if the journal files reside on devices such as disk or 3490s that do not support backward read.	Execute the sort utility and use as input either a set of archive journal files or the consolidated journal file produced in the preceding step. **Note:** For the sort parameters to use, see the ROLLBACK utility statement in the *CA IDMS Utilities Guide*.

Action	Steps
Remove from the restored files the effects of all updates made between the quiesce point and the end of the backup process. **When required:** Always	Execute the ROLLBACK utility statement specifying the HOTBACKUP option and using either the consolidated journal file or individual archive files concatenated in the sequence in which they were created. If the quiesce point for the affected areas does not coincide with the start of the input (or the end of the input if it was sorted in reverse sequence), use the STOP TIME parameter to identify the quiesce point. If STOP TIME is specified, also specify ACTIVE; otherwise specify ALL. If backward read is not supported for the device on which the input journal file resides, specify ROLLBACK3490 in the SYSIDMS parameter file associated with the ROLLBACK job step. This parameter is not necessary in a z/VM environment. If a consolidated journal file is not used as input in z/OS environments, specify the SORTED option.

Action	Steps
Consolidate, in the sequence in which they were created, the archive journal files created since the quiesce point established at the start of the backup procedure. **When required:** This step is necessary only under the following conditions: ■ In z/OS environments, if the subsequent ROLLFORWARD utility statement will be executed with the SEQUENTIAL option and more than one archive journal file must be processed. In a data sharing environment, if more than one member has updated the affected areas and the subsequent ROLLFORWARD utility statement will be executed with either the SEQUENTIAL or the ALL and STOP TIME options.	Execute one of the following and use as input the properly concatenated set of archive files: ■ FIX ARCHIVE utility statement ■ MERGE ARCHIVE utility statement ■ EXTRACT JOURNAL utility statement ■ Another comparable utility If consolidating archive files from multiple members and the subsequent rollforward will be executed with either the SEQUENTIAL or the ALL and STOP TIME parameters, use the MERGE ARCHIVE utility statement. **Note:** For more information, see Data Sharing Recovery Considerations (see page 666). If recovery also involves local mode journal files, the MERGE ARCHIVE utility statement can be used to consolidate local mode journal files and archive files. **Note:** For more information, see Recovery Procedures for Mixed-Mode Operations (see page 664).

Action	Steps
Reapply to the restored files all updates made since the quiesce point established at the beginning of the backup procedure. **When required:** Always.	Execute the ROLLFORWARD utility statement using either the consolidated journal file or individual archive files concatenated in the sequence in which they were created. If the journal files were consolidated using the EXTRACT JOURNAL utility, specify the FROM EXTRACT option. If FROM EXTRACT is not specified, then the following considerations apply: ■ Specify the SORTED option unless there is insufficient disk space available. SORTED must be specified if: ■ A consolidated journal file is not used as input in z/OS environments and more than one archive file must be processed. ■ The input journal file is on a device, such as a disk or a 3490 that does not support reading backwards. ■ Running ROLLFORWARD in a z/VM environment. If the SEQUENTIAL option is used and the quiesce point for the affected areas does not coincide with the start of the first input file, use the START TIME parameter to identify the quiesce point.

Restore Procedure 2

The use of this approach requires that:

- Two quiesce points were established during the hot backup procedure
- Backward read is supported for the input journal files. Backward read is not available in z/VM environments nor when the journal files reside on disk or a device such as a 3490

If either of these conditions are not satisfied, the first recovery approach must be followed.

Action	Steps
Copy the files that need to be recovered from the backup **When required:** Always.	Execute the RESTORE utility statement using the FILE option or another comparable utility.
Identify the two quiesce points that were taken during the backup process. Also identify the archive journal files that were created between those quiesce points and after the second quiesce point. **When required:** Always.	Use the PRINT JOURNAL utility statement, or if the quiesce point was established using the DCMT QUIESCE command, examine the operating system log for the DC/UCF system on which the DCMT command was issued.
Consolidate, in the sequence in which they were created, the archive journal files created between the two quiesce points established during the backup procedure. **When required:** This step is necessary only in z/OS, and data sharing environments if more than one archive journal file must be processed.	Execute one of the following and use as input the properly concatenated set of archive files: ■ FIX ARCHIVE utility statement ■ MERGE ARCHIVE utility statement ■ Another comparable utility **Note:** If consolidating archive files from multiple data sharing members, use the MERGE ARCHIVE utility statement. For more information, see Data Sharing Recovery Considerations. (see page 666)

Action	Steps
Reapply to the restored files all updates made between the two quiesce points.	Execute the ROLLFORWARD utility statement specifying the SEQUENTIAL option and using either the consolidated journal file or individual archive files concatenated in the sequence in which they were created. If the first quiesce point for the affected areas does not coincide with the start of the first input file, use the START TIME parameter to identify the quiesce point. If the second quiesce point does not coincide with the end of the last input file, use the STOP TIME parameter to identify the second quiesce point. **Note:** Output from the EXTRACT utility statement cannot be used to apply the images during this step.
Consolidate, in the sequence in which they were created, the archive journal files created after the second quiesce point established during the backup procedure. **When required:** This step is necessary only under the following conditions: ■ In z/OS environments, if the subsequent ROLLFORWARD utility statement will be executed with the SEQUENTIAL option and more than one archive journal file must be processed. ■ In a data sharing environment, if more than one member has updated the affected areas and the subsequent ROLLFORWARD utility statement will be executed with either the SEQUENTIAL or the ALL and STOP TIME options.	Execute one of the following and use as input the properly concatenated set of archive files: ■ FIX ARCHIVE utility statement ■ MERGE ARCHIVE utility statement ■ EXTRACT JOURNAL utility statement ■ Another comparable utility If consolidating archive files from multiple members and the subsequent rollforward will be executed with either the SEQUENTIAL or the ALL and STOP TIME parameters, use the MERGE ARCHIVE utility statement. **Note:** For more information, see Data Sharing Recovery Considerations (see page 666). If recovery also involves local mode journal files, the MERGE ARCHIVE utility statement can be used to consolidate local mode journal files and archive files. **Note:** For more information, see Recovery Procedures for Mixed-Mode Operations (see page 664).

Action	Steps
Reapply to the restored files, all updates made since the second quiesce point. **Note:** Updates made prior to the second quiesce point may also be reapplied during this step; however there is no need to do so. **When required:** Always.	Execute the ROLLFORWARD utility statement using either the consolidated journal file or individual archive files concatenated in the sequence in which they were created. If the journal files were consolidated using the EXTRACT JOURNAL utility, specify the FROM EXTRACT option. If FROM EXTRACT is not specified, then the following considerations apply: ■ Specify the SORTED option unless there is insufficient disk space available to perform the sort. SORTED must be specified if a consolidated journal file is not used as input in z/OS environments and more than one archive file must be processed. ■ If the SEQUENTIAL option is used and the quiesce point for the affected areas does not coincide with the start of the first input file, use the START TIME parameter to identify the quiesce point.

Reducing Recovery Time

Ways to Reduce Recovery Time

It is often critical to recover a database as quickly as possible to meet availability demands. The length of time it takes to recover can be reduced by:

- Limiting the scope of the recovery
- Reducing the time between backups
- Sorting journal images
- Pre-processing archive files

Limiting Scope of Recovery

One of the most significant factors affecting recovery time is the number of files being recovered. If recovering due to an I/O error, only a single file may need to be recovered. If recovering due to a journal I/O error, it may be necessary to recover all files in the database. To reduce time, recover only those files or areas impacted by the failure.

Reducing Time Between Backups

Another factor that affects recovery time is the number of journal images that must be applied to a restored file. One way to reduce the volume of journal images is to backup more frequently. Backups should be taken frequently enough that recovery times meet your operational requirements.

Sorting Journal Images

Another way to reduce the number of journal images applied to a restored file is to use the SORTED option of the ROLLFORWARD or ROLLBACK utility statement. By specifying this option, only the last AFTR image (in the case of ROLLFORWARD) or the first BFOR image (in the case of ROLLBACK) is applied to the database. While time and resources are required to sort the journal images, the number of I/Os to the database (and therefore the length of time needed to recover) may be significantly reduced using this option.

Note: There are restrictions on the use of the SORTED option when recovering from a hot backup. For more information, see 21.5.2, “Recovery From a Hot Backup”.

Preprocessing Archive Files

Another way to reduce the time needed to recover is to preprocess journal images using the EXTRACT JOURNAL utility statement. This utility eliminates redundant journal images by retaining only the last AFTR image for a dbkey. It creates an output file (called an extract file) that subsequently can be used as input to the ROLLFORWARD utility statement.

A backup plan may include the regular use of EXTRACT JOURNAL to pre-process archive journal files. If a recovery then becomes necessary, the extract files already exist and can be used in place of the original archive files to reduce the volume of journal images that must be applied to the database, thereby reducing the length of time it takes to recover.

To illustrate how this may be done, the EXTRACT JOURNAL utility might be executed each night. Its input would consist of all archive files produced since the previous night's extract or since the previous backup, whichever occurred most recently. If a recovery becomes necessary, the EXTRACT JOURNAL utility must be executed one more time to process the remaining archive files. After the database files are restored from the backup, the ROLLFORWARD utility is used to reapply updates. Its input is the concatenated set of extract files produced since the backup.

Note: There are restrictions on the use of extract files when recovering from a hot backup.

More Information

- For more information, see 21.5.2, “Recovery From a Hot Backup”.
- For more information about and for considerations in the use of the EXTRACT JOURNAL utility statement in a data sharing environment, see 21.11, “Data Sharing Recovery Considerations”.

Recovering a Large Number of Files

Operating System File Limitations

Some operating systems impose a limit on the number of files that can be accessed within a single job step. Except when exploiting extended file support in z/OS, the limit for a central version is the same as that for a batch job and so there are no special considerations involved in recovery.

Extended File Support

CA IDMS has extended the number of files that can be accessed by a central version in a z/OS operating system to exceed that which can be accessed by a batch job step. While useful, this capability may impact manual recovery.

Extended File Support and Manual Recovery

Under rare circumstances, it may be necessary to recover more files than can be accessed by a single batch job step. If this occurs, it will be necessary to split the recovery operation into multiple job steps each of which recovers a subset of the areas, files or segments within the DMCL. Each job step can access up to 3273 files.

Recovery Procedures After a Warmstart Failure

Before You Begin

Before you begin the recovery process, determine why the warmstart failed. Start by checking any shutdown or warmstart messages. The failure could be due to:

- Changes made to the DMCL or startup JCL
- Hardware problems
- Software maintenance
- Disabling change tracking

Corrective Action

If the failure is due to:

Change	Action
Changes in the DMCL and a timestamp mismatch is detected	Warmstart the system using the prior version of the DMCL load module **Note:** The warmstart failure could have been avoided through the use of change tracking by the CV.
Changes in the startup JCL	Correct the JCL and restart the system **Note:** The warmstart failure could have been avoided through the use of change tracking by the CV.
Software maintenance	Back out the maintenance and restart the system
Disabling change tracking	Warmstart the system specifying an IGNORE_SYSTRK_DMCL SYSIDMS parameter and ensure that the correct DMCL load module and startup JCL is used

Note: For more information about implementing change tracking, see "Change Tracking" in the *CA IDMS System Operations Guide*.

Steps

In the unlikely event that hardware or software problems prevent the warmstart process from recovering the database, follow these steps:

Action	Statement
Offload all journal files	ARCHIVE JOURNAL with the FULL option to offload all full journal files. This should be followed by an ARCHIVE JOURNAL with the READ option to offload the journal that was active when the abnormal system failure occurred.
Check for incomplete InDoubt transactions. Be sure to include all archive files created since the last quiesce point.	PRINT JOURNAL and FIX ARCHIVE
If incomplete InDoubt transactions exist, complete them manually by creating a new archive file. For more information, see 21.4.2, “Incomplete Transactions and Manual Recovery”.	FIX ARCHIVE using manual recovery control file input to complete the InDoubt transactions.
Recover the transactions that were active at the time of the system failure (that is, abended transactions)	ROLLBACK with the ACTIVE option
Unlock the areas that were not accessed during the rollback process. The ROLLBACK statement identifies what areas it unlocked.	UNLOCK
Reinitialize the journal files	FORMAT JOURNAL

Data Sharing Considerations

If a member of a data sharing group is unable to warmstart and manual recovery must be undertaken, any shared area that was being updated by the failing member must be quiesced in all other members of the data sharing group before the ROLLBACK utility is executed. To quiesce the area, change its status to OFFLINE or TRANSIENT RETRIEVAL. Do not use the DCMT QUIESCE command to quiesce the area.

Note: For additional data sharing considerations, see 21.11, "Data Sharing Recovery Considerations".

Recovery Procedures from Database File I/O Errors

What an I/O Error Means

An I/O error occurring on a database file indicates that an error occurred trying to read or write to the file. This may be caused by hardware malfunctions such as a channel problem, which if corrected, means that no recovery operation is needed. An I/O error can also be caused by a physically damaged file or disk device; this type of error requires recovery of the file.

Identifying a Database File I/O Error

When CA IDMS/DB encounters an I/O error in a database file, the following events occur:

1. CA IDMS/DB issues one of the following messages:
   - **DC205007**, which indicates a read error
   - **DC205008**, which indicates a write error
2. The transaction abends with a code of 3010 or 3011.
3. CA IDMS/DB performs automatic recovery processing.

If Recovery is Successful

If the recovery process is successful, CA IDMS/DB continues processing. To fix the I/O error, you must follow these steps:

Action	Statement
Take the area(s) associated with the bad database file offline	DCMT VARY AREA with the OFFLINE option
Identify the problem and fix it. If the problem is not associated with the database file itself (for example, the problem is due to a bad channel), perform step 3 after the problem is corrected; if the problem is due to a damaged file, perform the steps outlined for an unsuccessful recovery.	
Bring the area(s) associated with the database file online	DCMT VARY AREA with the ONLINE option

If the Recovery is Unsuccessful

If the recovery process is unsuccessful, CA IDMS/DB suspends the transaction and issues the following message:

```
DC205009 TRANSACTION SUSPENDED. TRANSACTION ID: transaction-id
```

When CA IDMS/DB issues this message, quiesce the area in which the problem occurred as quickly as possible to prevent additional transactions from readying the area. The following table identifies all the steps:

Action	Statement
Quiesce the affected area (see Considerations in this section)	DCMT VARY AREA with the TRANSIENT RETRIEVAL or OFFLINE options
Switch to a new journal file	DCMT VARY JOURNAL
De-allocate the file	DCMT VARY FILE with the DEALLOCATE option; use the FORCE option if the file cannot be closed (for example, because of a channel problem)
Restore a copy of the damaged file using the last backup tape as input. If the FORCE option was used in step 3, recreate the file with a new name	RESTORE with the FILE option
Rollforward the restored copy of the file using the archive journal files in the order they were created	Various. See 21.5, “Manual Recovery"
If the file was restored to a new location: ■ Recatalog it in z/OS ■ Update the standard labels in z/VSE	Operating system facilities
If the file was renamed in z/OS or z/VM, change its dataset name	DCMT VARY FILE with the DSNAME option
Make the new file available to the central version	DCMT VARY FILE with the ALLOCATE option
Re-activate the suspended transactions so they complete automatic recovery	DCMT VARY FILE with the ACTIVE option

Action	Statement
Re-activate the area for update processing	■ If the area was varied OFFLINE, issue DCMT VARY AREA with the ONLINE option ■ If the area was varied to TRANSIENT RETRIEVAL mode, first vary it OFFLINE and then ONLINE

Considerations

Quiescing the Area

Quiesce the area by varying it offline or retrieval. The differences are as follows:

- If the area is varied offline, no new transactions will be able to access the area until the recovery is complete and the area is varied online; existing transactions will complete if possible.
- If the area is varied to transient retrieval, transactions can continue to read data from the area but cannot update until the recovery is complete and the area is varied offline and back online. This may be useful if the area is mapped to many files (only one of which is damaged) or if only a small portion of the file is damaged. It can also be beneficial if most of the file blocks are in a buffer or a dataspace.

If the area to be recovered is a system area, it may be necessary to terminate predefined system run units by issuing a DCMT VARY RUN UNIT ... OFFLINE command to quiesce activity to the area. It is advisable to vary the status of a system area to transient retrieval rather than offline.

In a data sharing environment, it is important to quiesce a shared area in all members of the data sharing group. The broadcast capability of DCMT commands can be used to do this easily.

Renaming the File

If you restored the file under a new name, you must make sure that the correct file is used the next time the system is started. If change tracking is in effect for the DC/UCF system, CA IDMS automatically ensures that the correct file is used when the system is restarted following an abnormal termination. However, if change tracking is not in use or if you shut down the system, you must do one of the following:

- Rename (and recatalog) the restored file to its original name before restarting the DC/UCF system.
- Alter the system startup JCL to reference the new dataset name.
- After recovery is complete, modify the dataset name in the definition of the file, regenerate all DMCLs which include the file's segment and make the new DMCL available to the DC/UCF system.

If you fail to do one of the above, CA IDMS/DB will attempt to access the wrong file the next time the system is started. This may have serious consequences if the original file still exists.

More Information

- For more information about making a DMCL available to a runtime system, see see 7.13, “DMCL Statements”.
- For more information about implementing change tracking, see "Change Tracking" in the *CA IDMS System Operations Guide*.

Use of Deallocate Force

If the damaged file was de-allocated using the FORCE option, the DC/UCF system marks the file as closed and de-allocated but does not actually issue the corresponding operating system requests. For this reason, you must restore the file under a different dataset name. When the DC/UCF system is eventually shutdown, it will not shutdown successfully because the operating system will attempt to close the original file. This will either cause an abend or the DC/UCF system will hang. In either case, examine the messages produced on the log. If the following message appears, the database system has completed processing and no additional action is required:

```
DC200010 CA IDMS/DB Inactive
```

If this message does not appear, you should restart the system (after taking appropriate steps such as renaming the file) and then shut it down.

Correcting the Lock Option of an Area and File

If the area associated with a damaged database file is in retrieval mode or offline and the file was restored with the area lock on, then the area status is incompatible with the file status. If you try to vary the area online, IDMS responds with an error. To correct this situation, issue a DCMT VARY AREA command with the UPDATE LOCKED option. This command allows IDMS to vary the area to an update mode even though the file is locked.

InDoubt Transaction Considerations

No special action regarding InDoubt transactions should be necessary, since they will complete once the file is varied active and resynchronization takes place with the coordinator.

Recovery Procedures from Journal File I/O Errors

If an I/O error is encountered when accessing a journal file, the system responds differently depending on whether a read or write error is encountered:

- If a write error occurs, CA IDMS/DB swaps to a new journal file, re-issues the journal write and disables the use of the file on which the error occurred.
- If a read error occurs, CA IDMS/DB writes message DC205007 to the system log, indicating a read error and disables the journal file from further use.

The DC/UCF system will continue to operate without the use of the damaged journal file, although processing may be slower due to the availability of fewer journal files.

Automatic Recovery Failure

If a transaction abends (or issues a rollback) and, in order to recover, CA IDMS/DB must access a disabled journal file, it places the failing transaction in a suspended state and issues the following message to the log:

```
DC205009 Transaction suspended.  Transaction id xxxxxx
```

Recovery Procedure Steps

To recover from an I/O error on a journal file, follow these steps:

Action	Statement
1. Vary the affected journal file offline	DCMT VARY JOURNAL FILE OFFLINE
2. Monitor the status of the journal file within the system.	DCMT DISPLAY JOURNAL FILE

Action	Statement
3. If the journal file's status changes to OFFLINE, continue with Step 4. Otherwise, perform the steps outlined for unable to reach offline status.	
4. Identify the problem and correct it. If the problemis not associated with the journal file itself (the problem is due to a bad channel for example), correct the problem and continue with Step 5. If the problem is due to a damaged file, proceed with the steps outlined in "Repairing a Damaged Journal File."	
5 Vary the affected journal file online.	DCMT VARY JOURNAL FILE ONLINE

If the Journal Does Not Reach Offline Status

There are two conditions that may prevent a journal file from reaching an offline status:

- One or more active transactions are still dependent on the file for recovery
- The journal file has not been archived

If the journal file does not reach offline status, take the following actions:

Action	Statement
1. Determine if active transactions still depend on the journal file.	DCMT DISPLAY JOURNAL FILE PENDING TRANSACTIONS
2. If no pending transactions exist:	
2.1 Offload the journal file	ARCHIVE JOURNAL
2.2 If the archive is successful, the journal file will reach offline status, so proceed with Step 4 in the preceding table.	
2.3 If the archive is not successful, perform a quiescedbackup of all areas that were in update mode at the time of the I/O error and then proceed with the steps outlined in "Repairing a Damaged Journal File."	Various See "Quiesced Backup Procedure" in 21.2, “Backup Procedures".
3. If pending suspended transactions exist, quiesce all update activity within the system.	DCMT VARY AREA OR SEGMENT

Action	Statement
4. If pending non-suspended transactions exist:	
4.1 Wait for them to complete. Do not cancel them.	
4.2 If there are pending InDoubt distributed transactions, try to complete them by initiating resynchronization with their coordinator.	Various See 21.3.3, “Resynchronization”
5. If only pending suspended transactions exist, cancel the system and proceed with the steps outlined in "Manual Recovery Following a Journal File I/O Error."	Operating system facilities

Repairing a Damaged Journal File

Take the following actions to repair a damaged journal file while the DC/UCF system remains active:

Action	Statement
1. De-allocate the journal file	DCMT VARY JOURNAL FILE DEALLOCATE Use the FORCE option if the file cannot be closed (for example, because of a channel problem)
2. Allocate a new journal file; if the FORCE option was used in Step 1, create the file with a new name	Operating system facilities
3. Format the new journal file	FORMAT JOURNAL
4. If the file was allocated in a new location or with a new name ■ Recatalog it in z/OS Update the standard labels in z/VSE	Operating system facilities
5. If the file was allocated with a new name, make the name known to the DC/UCF system	DCMT VARY JOURNAL FILE DSNAME
6. Make the new file available to the DC/UCF system	DCMT VARY JOURNAL FILE ONLINE

Considerations for Renaming the File

If you allocated the new journal file using a new name, you must make sure that the correct file is used the next time the system is started. If change tracking is in effect for the DC/UCF system, CA IDMS automatically ensures that the correct file is used when the system is restarted following an abnormal termination. However, if change tracking is not in use or if you shutdown the system, you must do one of the following:

- Rename (and recatalog) the new journal file to the original name before restarting the DC/UCF system.
- Alter the system startup JCL to reference the new dataset name.
- If using dynamic allocation for journal files, after recovery is complete, modify the dataset name in the definition of the journal file, regenerate the impacted DMCL and make the new DMCL available to the DC/UCF system.

If you fail to do one of the above, CA IDMS attempts to access the wrong journal file the next time the system is started. This may have serious consequences if the original file still exists.

More Information

- For more information about making a DMCL available to a runtime system, see see 7.13, “DMCL Statements”.
- For more information about the use of change tracking, see "Change Tracking" in the *CA IDMS System Operations Guide*.

Manual Recovery Following a Journal File I/O Error

If one or more transactions cannot be rolled back due to their dependence on the damaged journal file, take the following actions to complete the recovery process:

Action	Statement
1. Restore all areas that were open at the time of the I/O error (including load and queue areas)	RESTORE
2. Check for incomplete InDoubt transactions using the archive journal files created since each backup was taken	PRINT JOURNAL or FIX ARCHIVE
3. If incomplete InDoubt transactions exist, complete them manually by creating a new archive file.	FIX ARCHIVE using manual recovery control file input to complete the InDoubt transactions
4. Roll forward all restored areas using the archive journal files created since each backup was taken or the corrected file created in the preceding step	ROLLFORWARD with the COMPLETED and AREA options

Action	Statement
5. Initialize all journal files	FORMAT JOURNAL with the ALL option
6. Backup all recovered database areas	BACKUP with the FILE option
7. Re-start the system	
8. Re-run all transactions that were not recovered	

Considerations

Quiescing System Activity

In a data sharing environment, it is important to quiesce a shared area in all members of the data sharing group. The broadcase capability of DCMT commands can be used to do this easily.

Conservative Approach

The steps outlined above take a conservative approach to the recovery process in two ways:

- No attempt is made to try and offload the damaged journal file. If the file can be offloaded, then the areas can be recovered using ROLLBACK (with the ACTIVE option) rather than using RESTORE and ROLLFORWARD.
- All areas being updated by the system are recovered. If you identify the areas that were being updated by the suspended transactions, then recovery can be limited to those areas and *other areas which are logically-associated.* To identify the areas that were being updated, you can use the DCMT DISPLAY TRANSACTION command for the suspended transactions or the DCMT DISPLAY AREA command which will identify the areas that have not quiesced.

Distributed Transaction Considerations

If journal information was lost due to the I/O error and areas had to be restored, any transactions whose journal images were lost were effectively backed out. This can lead to a mixed result for distributed transactions, since changes on other systems may have been committed. Unfortunately, there may be no way to determine what other systems are impacted due to the loss in journal information.

If you do know of another system that might be involved in one of these transactions, use their journal or log information to identify distributed transactions impacted by the failure. Look for distributed transactions in which the failed system was involved (as either a participant or a coordinator) and that were committed subsequent to the point to which an impacted area was restored.

Recovery Procedures for Local Mode Operations

Recovery procedures for local mode operations differ depending on whether you are journaling and if so, whether you are journaling to a disk device or tape device. The following topics provide the recovery procedures for each situation.

No Journaling

Use the Backup File

If you are not maintaining journal files during execution of a local mode job and the job terminates abnormally, you must restore all areas updated by the local mode application.

Journaling to a Tape Device

Steps

To recover a local mode database when journaling to a tape device, follow these steps:

Action	Statement
Rollback the database or areas of the database using as input the tape journal file created by the local mode job	■ If the job can be re-started from the last COMMIT point, useROLLBACK with the ACTIVE option ■ If the job has to be re-run from the beginning, use ROLLBACK with the ALL option
Re-run the application	

Journaling to a Disk Device

Steps

If you are journaling a local mode job to a disk device, follow these steps:

Action	Statement
Copy the journal file to a tape device	Operating system utility
Follow the steps outlined above for journaling to a tape device above	

Using an Incomplete Journal File

What is an Incomplete Journal File?

An incomplete journal file is a journal file that does not contain a final ABRT checkpoint for the active transaction or even an end-of-file mark. This occurs when the journal file has been unexpectedly interrupted, for example, when the operating system crashes. An incomplete journal file is not suitable for recovering your database. To make a suitable journal file for recovery, use the FIX ARCHIVE utility statement, which:

- Reads the damaged file and creates a new one
- Writes an ABRT checkpoint at the end of the new file

Steps

To recover a database in local mode, using an incomplete journal file, follow these steps:

Action	Statement
Fix the journal file	FIX ARCHIVE
Recover the database using the output from the FIX ARCHIVE utility statement as described for journaling to a tape device above.	

Recovery Procedures for Mixed-Mode Operations

What is a Mixed-Mode Operation?

When database areas have been updated both in local mode and under the central version (for example, when an area has been varied offline, subsequently updated by a local transaction that used journaling, and then varied back online), the database must be restored by using both the local and the central version journals.

Mixed Mode Recovery

The following scenario is an example of synchronizing the recovery operations by explicitly using both the central version and local journals to ensure proper recovery of all database areas:

```
6 a.m.        Nightly backups taken

8 a.m.        System startup:  AREA1, AREA2,
              AREA3 are readied in update
              mode under the central version.

10:30 a.m.    AREA1 is varied offline.
              While offline, a local mode program
              (using a tape journal) updates
              AREA1 while the central version
              continues to update AREA2 and
              AREA3.

11:30 a.m.    A VARY JOURNAL command is issued
              for the central version journal.
              AREA1 is varied back online and
              the central version continues to
              update AREA1, AREA2, and AREA3.

12:00 p.m.    Database file I/O error occurs
              on AREA1.
```

When the database file I/O error occurs, the affected file associated with AREA1 must be restored by using both the local and central version journals.

Steps to Recover the Database

The following steps illustrate one approach to recovery, given the situation outlined above. Note that with this approach two separate rollforward operations are used. To process journal images from both central version and local mode operations in a single execution of the ROLLFORWARD utility, you must use the alternate recovery approach described next.

Action	Statement
Restore the damaged file using the backup tapes produced at 6 a.m.	RESTORE with the FILE option
Rollforward all archive files produced before 11:30	ROLLFORWARD FILE specifying ALL
Rollforward the local journal file, restoring the file up to 11:30 a.m.	ROLLFORWARD FILE specifying ALL
Rollforward using the archive files produced between 11:30 a.m. and 12:00 p.m.	ROLLFORWARD FILE with the COMPLETE option

For a complete description of the recovery process, see 21.7, “Recovery Recovery Procedures from Database File I/O Errors".

An Alternate Approach

The following steps illustrate an alternate approach to recovery in a mixed-mode environment. With this approach, the local mode journal file is first merged with the archive files produced by the central version and the merged output file is used to recover the database in a single rollforward operation.

Action	Statement
Restore the damaged file using the backup tapes produced at 6 a.m.	RESTORE with the FILE option.
Merge the local mode journal file with all archive files produced since 8 a.m.	MERGE ARCHIVE specifying the COMPLETE option

For a complete description of the recovery process, see 21.7, "Recovery Procedures from Database File I/O Errors".

Data Sharing Recovery Considerations

Quiescing Update Activity

Whenever it becomes necessary to quiesce access to an area during a recovery operation, the quiesce must apply to all members of a data sharing group. For recovery purposes, the quiesce will usually be done by varying the area status to OFFLINE or TRANSIENT RETRIEVAL using a DCMT VARY command. This command must be executed in every member to establish a group-wide quiesce for a shared area. To do this easily, the command may be broadcast to other members of the group.

Occasionally, it will be sufficient to quiesce update access to an area through the use of a DCMT QUIESCE command. This command will automatically propagate the quiesce for a shared area to all group members, so there is no need to execute it on more than one member of the group.

Note: For more information about DCMT commands and how to broadcast them, see the *CA IDMS System Tasks and Operator Commands Guide*.

Recovery from a Warmstart Failure

If warmstart fails for one or more members of a data sharing group, recovery can proceed just as if the DC/UCF systems were not data sharing members. This is true even if manual recovery is necessary provided that all shared areas being updated by the members at the time of failure are quiesced in the remaining members of the group. The quiesce should be done by varying the status of the affected areas to OFFLINE using a DCMT VARY command.

If manual recovery is necessary, each member can be recovered independently provided that:

- Only the ROLLBACK recovery utility is used
- The ACTIVE option is specified when executing ROLLBACK, and
- Executions of the ROLLBACK utility are serialized

Failure to comply with these conditions, may result in database corruption.

Recovery from Other Types of Failures

Except when following the above procedure to recover from a warmstart failure, the archive files from all members that have updated a shared area since the backup was taken must be included in any manual recovery. Furthermore, the journal images from all members must be processed together in the same execution of the ROLLFORWARD or ROLLBACK utility. It is not valid to process the images from one member in one execution followed by the images from another member in another execution, since journal images must be processed in chronological sequence.

Recovery from a warmstart failure is an exception to this rule only because records updated by one member cannot be accessed by another member until the changes are committed or rolled out. If warmstart fails, the unrecovered records remain locked, so no other member can update them. This means that there will never be more than a single member with a before image for an unrecovered record and so inter-member sequencing is not important.

Using MERGE ARCHIVE

The MERGE ARCHIVE utility is used to merge the journal images from multiple members so that they are in chronological sequence. As noted above, most recovery utilities require that journal images be processed chronologically. In a data sharing environment, the journal images produced by each member are in chronological sequence, but the images for areas concurrently updated by multiple members are contained in each member's archive files. The MERGE ARCHIVE utility interleaves the journal images from multiple members so that they occur in date/time sequence. The resulting output file may then be used as input to a ROLLFORWARD, ROLLBACK, or EXTRACT JOURNAL utility statement.

When executing the MERGE ARCHIVE utility statement, the input consists of a concatenated set of archive files and optionally a merge archive file produced from a previous execution of the MERGE ARCHIVE utility. Archive files produced by a single member must be processed in the order in which they were created. Archive files from different members may be processed in any order relative to those of other members.

When to Use MERGE ARCHIVE

The output of the MERGE ARCHIVE utility can always be used as input to the ROLLFORWARD, ROLLBACK, and EXTRACT JOURNAL utility statements in place of the original archive files. It can also be used to combine local mode journal files and archive files when mixed-mode updates must be recovered.

However, while optional in most cases, MERGE ARCHIVE **must** be used to merge the journal images of multiple data sharing group members before those images are processed by:

- ROLLFORWARD or ROLLBACK utility statements that specify the SEQUENTIAL option.
- ROLLFORWARD, ROLLBACK, or EXTRACT JOURNAL utility statements that specify both the ALL and STOP TIME options.

Using EXTRACT JOURNAL

The EXTRACT JOURNAL utility is used to preprocess journal images in order to reduce recovery time. This utility can also be used in a data sharing environment. Any of the following are valid approaches to its use:

- Separately preprocess the archive files of each member
- Preprocess the archive files of multiple members together
- Merge the archive files of multiple members using the MERGE ARCHIVE utility and then preprocess the merge file

You must use the third approach if the ALL and STOP TIME parameters are specified on the EXTRACT JOURNAL utility statement; otherwise, any of the above approaches can be used to preprocess journal files in a data sharing environment.

If using either of the first two approaches, the EXTRACT JOURNAL utility statement can be executed on a periodic basis to preprocess the archive files created since its previous execution, or since a backup was taken. If recovery becomes necessary, all extract files produced since the backup must be concatenated as input to a single execution of the ROLLFORWARD utility. The order in which the extract files are concatenated must be such that the journal images for each member are in chronological sequence. It makes no difference in which order the images of one member occur in relation to those of another member.

If using the third approach, the entire set of archive files produced by group members that have updated the affected areas must be merged prior to executing the EXTRACT JOURNAL utility. The MERGE ARCHIVE utility can be executed on a periodic basis to merge the archive files created since its previous execution with the previously created merge file. The EXTRACT JOURNAL utility can then be used to preprocess the final merge file.

Note: For more information about executing both the MERGE ARCHIVE and the EXTRACT JOURNAL utility statements, see the *CA IDMS Utilities Guide*.

Coupling Facility Failures

Certain types of failures are unique to a data sharing environment, such as the loss of a coupling facility or a structure within the coupling facility. In some cases, all members of a group will fail and recovery must be coordinated across the group, a process called "group restart."

Note: For more information about recovering from coupling facility failures and group restart, see the *CA IDMS System Operations Guide*.

Considerations for Recovery of Native VSAM Files

About Recovery for Native VSAM Files

CA IDMS/DB performs journaling for native VSAM files just like it does for other types of files it supports. The recovery procedures described in this chapter apply to native VSAM files also. The processing difference is that the BACKUP and RESTORE utility statements cannot be used with native VSAM files. Instead, use IDCAMS or some other utility for backing up and restoring the file.

Potential Problems

Since VSAM controls the actual updating of the data sets, recovery problems may occur. If a total system failure occurs after CA IDMS/DB passes control to VSAM, automatic recovery is not guaranteed. Therefore, you should back up native VSAM data sets frequently, as described in the appropriate VSAM documentation. Recovery can then be accomplished by restoring your file Using IDCAMS (or some other utility) and ROLLFORWARD utility statements.

File Verification After Failure

If a DC/UCF system fails or a local mode application terminates abnormally, you must issue the IDCAMS VERIFY command for native VSAM files that were open for update at the time of the failure.

Limitations for ESDS Areas

You cannot use the ROLLBACK utility statement for an ESDS area to which a record has been added, because VSAM does not allow the necessary erase.

Limitations for KSDS Areas

Due to limitations within the VSAM access method, ROLLFORWARD and ROLLBACK cannot be run with the SORTED option to recover native VSAM KSDS areas. If you need to use the SORTED option, because of the volume of data, and a database that contains a mixture of KSDS, ESDS, and/or RRDS native VSAM files, follow these steps:

Action	Statement
Restore the native VSAM files	Operating system facility
Rollforward or rollback the area that maps to the KSDS file; the utility statement recovers the KSDS file and any associated alternate indexes.	ROLLFORWARD or ROLLBACK with the SEQUENTIAL option
Rollforward or rollback all the remaining areas or files.	ROLLFORWARD or ROLLBACK with the SORTED option

Chapter 22: Loading a Non-SQL Defined Database

This section contains the following topics:

Database Loading

Loading Options

To load a database defined with non-SQL DDL statements, you can use either:

- The FASTLOAD utility statement
- A user-written load program

FASTLOAD Utility Statement

To use FASTLOAD, you must write and compile a format program that specifies how to load the data. After executing the format program, you invoke the FASTLOAD utility statement, which loads record occurrences into the database and makes set connections using the output from the format program. It also builds indexes during the load process.

User-Written Program

You can also load a database by using DML commands in a user-written application. The application can be written in any of the languages CA IDMS/DB supports.

If you use a user-written program to load the database, you should organize the record occurrences in the input file so that they mimic the structure of the database. For example, you should sort the records so that a CALC record is followed by its VIA member record occurrences. Steps for organizing the input file appear in more detail later in this chapter.

Advantages of FASTLOAD

FASTLOAD is often more efficient than a user-written program for loading a database with complicated structures (for example, multiple-member sets or multi-level record relationships). Additionally, FASTLOAD does not require pre-sorted data. As part of its internal processing, FASTLOAD sorts the data at certain points during the load process.

Loading Database Records Using FASTLOAD

Requires a User-Written Format Program

To use FASTLOAD, you must write a format program that uses subroutines provided by CA to prepare data for input to the FASTLOAD utility statement.

Note: For more information about and a description of the format program, see the FASTLOAD statement in the *CA IDMS Utilities Guide*.

General Considerations

Always Load in Local Mode

You must load a database in local mode. Journaling is not required, and is not recommended when loading a database for these reasons:

- The load utility does not maintain checkpoints
- It's easier to re-run a failed job step than to recover the database
- Journaling can impact performance

Cross-Area Sets

- If the owner and member records of an automatic set exist in different database areas, load the areas together.
- If the owner and member records of a manual set exist in different database areas, either:
  - Load the areas together
  - Run a user-written program to connect the records after loading the entire database

CALC Records

The target page for CALC records to be loaded into a database can be determined in one of two ways:

- By the standard CA IDMS/DB CALC routine (IDMSCALC).
- By a user-written CALC exit routine (IDMSCLCX) that was compiled and link-edited with IDMSUXIT.

  Important: If you determine the target page using IDMSCALC, you must use it whenever the database is accessed; likewise, if you use the IDMSCLCX user exit, you must continue to use an IDMSUXIT module with which it is linked.

Note: For more information on enabling user exits by linking IDMSUXIT, refer to the "User Exits" section of the *CA IDMS Systems Operations Guide*.

Compressed Data

If the schema definition specifies compression for a record type, CA IDMS/DB compresses the record before it stores it during a load operation. Therefore, before you begin the load procedure, be sure the schema definition includes the information it requires to compress the record occurrences.

Reserving Space on the Page

To reserve space for the storage of additional records on a page or for increases in the length of records stored on a page, add an area override to the DMCL that specifies a page reserve. When the load is complete, you can remove the area override.

Buffers

The DMCL that you use to load the database should contain a local mode buffer that contains at least 10 pages. One large buffer should be sufficient. However, you may obtain performance improvements by assigning the files associated with each area to a separate buffer. If you don't have enough resources, then try to assign the files associated with the following areas to separate buffers:

- Index area
- Areas for which the owner record exists in one area and the member record exists in another area

Considerations for Large Databases

A large database should be loaded in portions. The FASTLOAD statement assumes that all record occurrences that are connected by automatic sets will be loaded at the same time. For a large database, this assumption can be limiting. To load a large database:

1. Group the record types so that there are not automatic sets between the groups
2. Load each group of record types
3. If manual set connections exist between records in different groups, connect the records by executing a user-written program

Subschema Requirements

The subschema that you use in the load process must:

- Include all records being loaded and all set relationships in which the records participate
- Allow all affected areas to be readied in exclusive update mode

FASTLOAD Procedure

Steps

To load a database for the first time, follow these steps:

1. Write and compile a format program that specifies how to load the data. For more information about the format program, see the *CA IDMS Utilities Guide*
2. Link-edit the format program with IDMSDBLU
3. Define the segments, areas, and files that represent the physical database
4. Add the segment definition to the DMCL and make the DMCL available to the runtime environment
5. Format the database files to be loaded using the FORMAT utility statement with the FILE option
6. Execute the format program
7. Load the database using the output from the format program as the input to FASTLOAD
8. Back up the database areas using the BACKUP utility statement or any comparable backup utility
9. Verify the validity of the loaded database using:
   - IDMSDBAN, to verify the physical integrity of the database
   - CA OLQ, CA Culprit, or some other retrieval job to verify the data in the database

Loading Database Records Using a User-Written Program

Before you load a database using a user-written program, you must first organize the data in the input file. This section Discusses how to organize the record occurrences followed by the procedure to load the database.

Organizing Input Data for a User-Written Program

Organize Record Occurrences to Match Schema

To make the database load as efficient as possible, you need to organize the record occurrences to match the structure of the database. For example, you want a CALC owner record to be followed by its VIA member records. The discussion below identifies how to organize the data.

Step 1: Identify the Record Types

The first step in organizing input data is to identify the type of each record. To identify the type of record, add the record's ID to the beginning of each record occurrence. For example, the ID of the DEPARTMENT record is 410; the ID of the EMPLOYEE record is 415.

Step 2: Identify CALC Clusters

A CALC cluster is an occurrence of a CALC record, all of its VIA member records, and all VIA member records of a VIA member record occurrence. For efficient database processing, all the records within a CALC cluster should fit on one page (and thereby, can be processed with one I/O). If the records do not fit on one page, then store the most frequently accessed record types immediately following the CALC record occurrence so that they have a better chance of being stored on the same page as the owner.

Step 3: Form CALC Cluster Hierarchies

A hierarchy is a collection of CALC clusters. For example, if a CALC record occurrence in one cluster is owned by a record in another cluster, you have a hierarchy of CALC clusters. In the Commonweather database, both the OFFICE and DEPARTMENT records own occurrences of the EMPLOYEE record, which in turn owns VIA member record occurrences. In deciding what records to include in the CALC cluster hierarchy, consider the number of CALC record occurrences. For example, if the DEPARTMENT record has many more occurrences then the OFFICE record, then store the EMPLOYEE records immediately after the owning DEPARTMENT record. This potentially saves an I/O because you won't need to reestablish currency on the DEPARTMENT record occurrence later on.

Hierarchies are loaded from top-to-bottom, left-to-right order. When you store the owner of a CALC cluster, you establish currency to store the member of a CALC cluster.

Step 4: Sort the Records in a Hierarchy

To sort records within a hierarchy, add a prefix to the beginning of the record occurrence. The prefix contains the record id and sequence number for each level of the hierarchy. For example, the DEPARTMENT, EMPLOYEE, EMPOSITION record hierarchy might have a prefix that looks like this:

ID and sequence number of each level in hierarchy			Record ID	Record Occurrence
410/1	0/0	0/0	410	Department record 1
410/1	415/1	0/0	415	Employee record 1
410/1	415/1	420/1	420	Emposition record 1
410/1	415/1	420/2	420	Emposition record 2

Step 5: Order the Occurrences of Each Hierarchy

A database page will typically hold more than one database cluster. Therefore, you can load multiple clusters with one I/O if you load all the hierarchies that target to the same database page. To sort the hierarchy occurrences, add the CALC target page number of the top cluster in the hierarchy to the beginning of the input record.

Note: To determine the CALC target page, use IDMSCALC in the program that creates the input file; for more information about IDMSCALC, see the *CA IDMS Utilities Guide*.

Step 6: Include Records Excluded from the Hierarchies

Some records do not fall within a hierarchy. For example, suppose you did not include the OFFICE record, which owns EMPLOYEE record occurrences in a CALC cluster hierarchy. To load owner records that fall outside of a hierarchy:

1. Position the non-VIA owner records at the beginning of the input file, before any records that form part of a hierarchy, by adding an identifier to the beginning of each input record. For example, the identifier of the OFFICE record type might be 4 and the identifier of the DEPARTMENT, EMPLOYEE, EMPOSITION hierarchy might be 5.

2. Add the key of the non-VIA owner record to the end of the hierarchy record occurrence; at load time, use the key to find the owner before storing the member. For example, add the OFFICE-CODE-0450 field to the end of each EMPLOYEE record occurrence.

Step 7: Order Sorted and Indexed Sets

Sorted sets should always be loaded in the same order as the sort sequence. To sort the input data:

- For a set within a hierarchy, replace the sequence number field at the record's level in the hierarchy with the sort key of the set; for example, if the EMP-EMPOSITION set is a sorted set, replace the sequence number for occurrences of the EMPOSITION record with the record's sort key in the prefix portion of the input record.
- For a set outside of a hierarchy, follow these steps:
    1. Re-define the set as manual
    2. Create a file containing records with these fields: the owner's page, the set name, the owner's CALC key, the set's sort key, the dbkey of the member record
    3. Sort the file in:
        - Descending order by page
        - Ascending order by set name and owner key
        - Either ascending or descending order by sort key
    4. After loading the database, connect the set members using a user-written program

Step 8: Sort the Input Records

Sort the input records in:

- Ascending order by identifier
- Descending order by target page number
- Ascending order by the concatenation of all ID and sequence fields that represent a hierarchy

Note: If records are to be stored VIA a system-level index, they should be sorted in the reverse order of their VIA index so records at the end of the index will be processed first by the user-written format program. This ensures that the physical sequence of the records on the database matches the sequence of the index.

Loading the Database

To load a database for the first time using a user-written program, follow these steps:

Action	Statement
Write and compile a load program that specifies how to load the data.	
Optionally, tailor the DMCL to be used for the load operation	ALTER DMCL
If altered, make the DMCL available to the local mode runtime environment	See Chapter 5, "Defining, Generating, and Punching a DMCL"
Format the database files to be loaded	FORMAT
Load the database using as input the sorted input file	Execute the user-written program
If necessary, connect members to sets treated as manual during the load	Execute the user-written program
Back up the database areas	BACKUP or any comparable backup utility
Verify the validity of the loaded database	CA OLQ, CA Culprit, or some other retrieval job to verify the data in the database

Example

The following example shows code to load the DEPARTMENT hierarchy:

```
read an input record
repeat until end-of-file
   if record-id = 410
         move DEPARTMENT record
         store DEPARTMENT
   else if record-id = 415
         move OFFICE key
         find calc OFFICE
         move EMPLOYEE record
         store EMPLOYEE
   else if record-id = 420
         move JOB key
         find calc job
         move EMPOSITION record
         store EMPOSITION
   else if record-id = 425
         move SKILL key
         find calc SKILL
         move EXPERTISE record
         store EXPERTISE
   end-if
read next input record
end-repeat
```

More Information

- For more information about utility statements mentioned, see the *CA IDMS Utilities Guide*.
- For more information about loading an SQL-defined database, see Chapter 23, “Loading an SQL-Defined Database”.
- For more information about the IDMSCLCX user exit, see the *CA IDMS System Operations Guide*.

Chapter 23: Loading an SQL-Defined Database

This section contains the following topics:

Database Loading

Loading SQL Defined Databases

CA IDMS provides utilities to efficiently load a database that has been defined with SQL DDL statements. The entire load operation can be performed as a single operation using the LOAD utility statement, or it can be executed as separate operations using a combination of the LOAD, BUILD, and VALIDATE utility statements. Regardless of which method is used, loading an SQL-defined database consists of multiple phases and steps within those phases.

Loading Phases

The following table summarizes the phases involved with loading an SQL-defined database. The load process was designed to accommodate both small databases and very large databases and allow flexibility in tailoring the load process to the characteristics of the data being loaded:

Phase	What it does
Load	Loads the specified tables
Build	Builds indexes and linked index constraints for the specified tables; this phase can be bypassed if neither linked index constraints nor non-clustering indexes are defined on the specified tables
Validate	Validates referential constraints in which the specified tables participate

Steps Within Phases

Each of these phases, in turn, is composed of sub-phases called *steps*. The following table summarizes the function of each step:

Phase	Step	What it does
Load	Step 1	Processes data in preparation for sorting; this step can be bypassed if data is already sorted
	Step 2	■ Loads the table rows ■ Connects linked, clustered constraints ■ Builds clustering indexes
Build	Step 1	Performs an area sweep in the absence of an intermediate extract file
	Step 2	Finds the db-keys of rows that participate in the referenced table of a linked index referential constraint
	Step 3	Builds non-clustering indexes and linked indexes
	Step 4	Updates the prefixes of rows that participate as the referencing table of a linked index referential constraint
Validate	Step 1	Validates only those constraints that can be processed efficiently in a single pass and extracts information about other referential constraints
	Step 2	Validates any referential constraints bypassed in Step 1

Loading Options

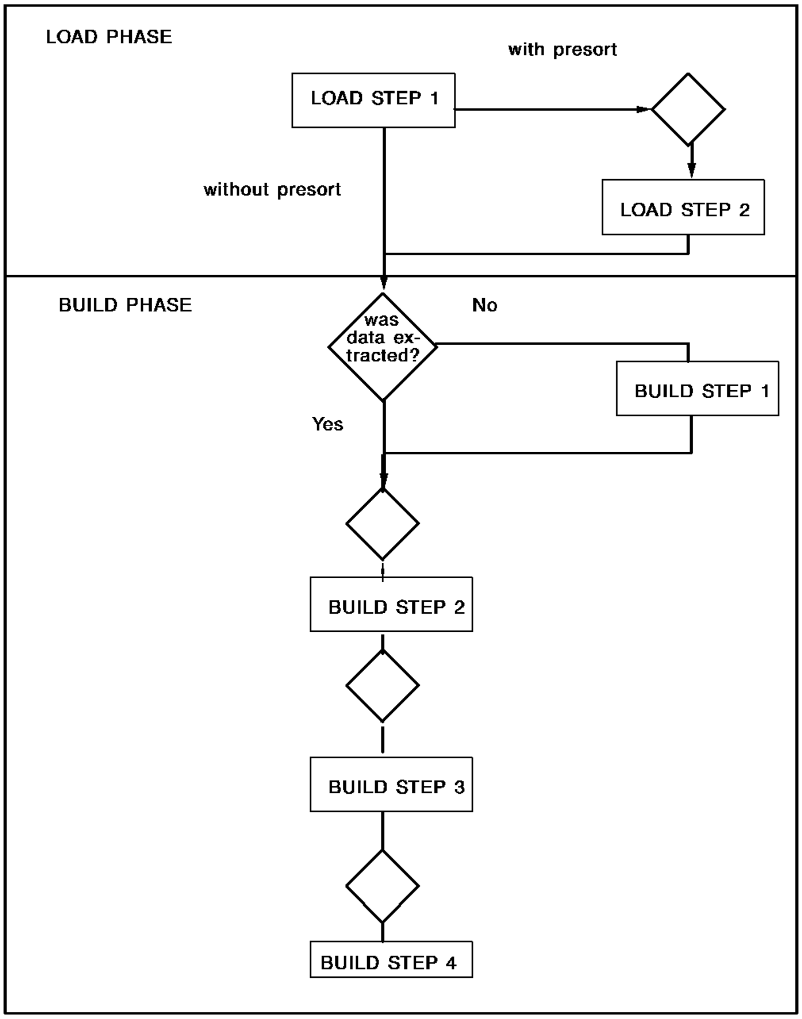
CA IDMS/DB offers you the following loading options:

Option	Description	When to use it
Full load	Loads, builds and validates the specified tables	Always, unless special considerations apply
Phased load	Executes each phase (load, build, and validate) separately	When loading a number of tables one at a time or in groups; defer build and validate phases until all the tables have been loaded
Segmented load	Loads portions of input in separate operations	When loading extremely large tables; defer the build and validate steps until all the input records have been processed

Option	Description	When to use it
Stepped load	Executes each step of a phase (load, build, and validate) separately	When loading extremely large tables for which external sort packages may be more efficient or when space for intermediate work files or tape drives is at a premium

Load Flow Diagram

The following diagram illustrates the load and build phases of the process described above:



CA IDMS/DB Enforces All Constraints During the Load

CA IDMS/DB enforces all constraints during the load process. That is, it enforces:

- Referential constraints
- Data type constraints
- Check constraints
- Unique constraints

For example, if a table allows only specified values to be stored in a column, CA IDMS/DB stores only valid values. CA IDMS/DB also assigns default values for columns for which no input values are supplied, provided the column was defined to allow null or default values.

Loading Considerations

Loading Multiple Tables

If you are loading multiple tables, it may be necessary to split the process into separate load operations and process them in a certain order. Use these rules to determine whether this is necessary and the correct sequence in which to perform the load operations.

- Tables clustered through a linked or unlinked constraint cannot be loaded until the referenced table is loaded and any index on the referenced key has been built.
- Linked index constraints cannot be built until the referenced table is loaded and any index on the referenced key has been built.

Using Pre-Sorted Data

Before CA IDMS/DB loads data, it sorts the data using a sort sequence best suited to the table's characteristics. If you have already sorted the input data, you can tell CA IDMS/DB to skip the sort phase.

Providing Sorted Data

To sort the data yourself, follow these recommendations to achieve the most efficient load for your tables:

Table Characteristic	Recommended Sort Sequence
Table has a clustered index	Sort on index key
Table has a clustered referential constraint	Sort on foreign key of the referencing table

Table Characteristic	Recommended Sort Sequence
Table has a CALC key	Sort on CALC-key target page; to do this, use the IDMSCALC utility program to determine the target page and append the target page to the input record

Database Buffers Used During Load

You must load a database in local mode. The DMCL that you use for the load should specify buffers for the areas being loaded that contain at least 10 pages. The larger the buffer, the more efficient the load.

Reserving Space on the Page

If you want to leave free space on the database pages following the load, add an area override in the DMCL that specifies a page reserve. After the load is complete, remove the area override so that new rows and index entries can use the free space. This technique is especially useful for areas that contain only indexes or that contain tables clustered on an index.

Error Handling

CA IDMS/DB may encounter errors during each phase of the load process. You can instruct CA IDMS/DB what to do in response to these errors, for example, to continue processing or to quit following a specified number of errors. The following table summarizes the types of errors that can occur within each phase:

Phase	Type of Error	Corrective Action
All phases	Table not defined in the catalog	Define the table in the catalog
Load	■ Check constraint violation ■ Invalid data values ■ Unique constraint violation on a CALC key,clustering index, or linked clustered constraint	No corrective action needed; however, row is not inserted and subsequent build and validate phases may fail.
	■ Referential constraint violation on a linked clustered constraint	Ensure that the referenced table has been loaded and any referenced key index built prior to loading the referencing table.

Phase	Type of Error	Corrective Action
Build	■ Unique constraint violation on non-clustering indexor linked index constraint ■ Referential constraint violation on a linked index constraint	FIX PAGE utility statement or reload data
Validate	Invalid referential constraint	■ INSERT to store missing owner ■ UPDATE to change invalid foreign key ■ DELETE to remove invalid referencing rows

Input Data Used in the Build Phase

You can enter the BUILD phase of the load process using data stored in intermediate work files created by the LOAD phase or by instructing CA IDMS/DB to extract the necessary information as the first step in the build process. Intermediate work files are generally used when you intend to enter the BUILD phase immediately following the LOAD phase; typically, you instruct CA IDMS/DB to extract the information if some time elapses between the two phases.

The following table summarizes how to specify these options:

BUILD Phase Input	Load and Build Option	LOAD Statement	BUILD Statement
Intermediate work file	Phased load and build	LOAD NO VALIDATE	None
	Stepped load	LOAD STEP1 EXTRACT	Start with BUILD STEP2
Extracted work file	Phased load	LOAD NO BUILD	Start with BUILD STEP1

Enhancing Load Performance

The following list identifies some ways to enhance the performance of your load operations:

- If possible, load several tables at the same time
- Validate several tables at once
- Always load using sorted data; either letting CA IDMS/DB sort the data or by supplying pre-sorted data
- Increase the number of pages in the buffer(s)

Contents of the Input File

Mixed Input Records

The input file to the load process can contain different types of input records. For example, the input file might contain an EMPLOYEE record, followed by a DEPARTMENT record, followed by an OFFICE record and so on; to distinguish the different types of records, you must include a record identifier (in this example, at the end of the record):

```
0574SMITH    JOHN   254 WILLOW ST  NEEDHAM  MA   4035 415
4001PERSONNEL          MASON     PAULA    5538   0020 410
0020CICAGO  3 CORPORATE PLACE                        450
```

Note: By including record identifiers at the end of the input records, you may be able to avoid listing individual column definitions in the LOAD statement.

Loading Multiple Tables

You can load more than one table in the same load operation by using one of the following techniques:

- By specifying selection criteria applied against records in the input file. For example, to load the EMPLOYEE table, using the example above, you could select all input records with value '415' as a record identifier.
- By selecting specific fields from one input record that contains data pertinent to multiple tables. For example, the input record may contain values to be stored in table EMPLOYEE and values to be stored in table DEPARTMENT.

Identifying Columns Implicitly

If, in the LOAD statement, you do not explicitly list the columns in the table to be loaded, CA IDMS/DB assumes that values are supplied for all columns in the table. It starts with position 1 of the input record and extracts input values for each column of the table. To be successful, the input data must match the order, data type, length, and null criteria specified in the table definition. Columns that allow null values must be represented by a data field and an indicator field, which is described under "Null values".

Identifying Columns Explicitly

If you supply values for only some of the columns within the table or if the order or data types of the values in the input file do not match that of the columns in the table, you must tell CA IDMS/DB:

- Where to find the column values in the input record by specifying their start position relative to the beginning of the input record
- The data type of the input record value
- The null value criteria for input values, if applicable

If you omit a column name, the column must either:

- Allow null values
- Allow a default value

Data Types

If you explicitly list the columns to be loaded, the data type of the value to be stored can be different from the data type defined for the column provided the data types are compatible. For example, a column defined as CHARACTER is compatible with data types VARCHAR, DATE, TIME, and TIMESTAMP.

Note: For more information about compatible data types, see the *CA IDMS SQL Reference Guide*.

Null Values

Null values in an input file can be represented as either:

- A specific data value, designated by you in the NULL IF clause of the LOAD statement.
- An indicator field, immediately following the data field. This field is either a 1, 2, or 4 byte binary field and must contain a value of:
  - 0, to indicate a non-null data value
  - -1, to indicate a null value

  If you do not explicitly list the columns to be loaded, then all columns that permit null values must be followed by a *4-byte* indicator field.

Loading Procedures

The remainder of this chapter provides procedures and examples for:

- Steps that apply to all load procedures
- A full load procedure
- A phased load procedure
- A segmented load procedure
- A stepped load procedure

Note: Only one LOAD, BUILD, or VALIDATE statement may be performed during one execution of the batch command facility; for example, you cannot submit two LOAD statements at one time.

Steps That Apply to All Load Procedures

Steps Before the Load

Regardless of what load procedure you use, perform the following actions *before* loading the data:

Action	Statement
Define the tables to be loaded	CREATE TABLE
Create the input file or files of data to be loaded using CA Culprit, CA OLQ (batch), or a user-written program	
Vary the areas in which the tables reside offline to DC/UCF	BACKUP or a comparable backup utility

Steps After the Load

After loading the data, perform these steps:

Action	Statement
Optionally, verify the result by retrieving data from the loaded tables	SELECT statements
Back up the areas in which the tables reside	BACKUP or a comparable backup utility
Vary the areas online	DCMT VARY AREA with the ONLINE option

Full Load Procedure

Steps

Follow these steps to perform a full load of an SQL-defined database:

Action	Statement
In local mode, load, build, and validate one or more database tables	LOAD

Example

This example loads, builds, and validates tables ASSIGNMENT, CONSULTANT, EXPERTISE, SKILL, and PROJECT. Each of these tables is independent of those in other areas of the EMPLOYEE database. By default, CA IDMS/DB sorts the input data before it loads the tables. Also by default, if it finds any errors during any phase of the load procedure, it stops.

To load each table, CA IDMS/DB applies selection criteria against each input record it reads. For example, the ASSIGNMENT table receives all input records where the value in position 210 of the input record equals '512'. Similarly, the EXPERTISE table receives all input records where the value in position 210 equals '320'.

```
load
  into demoproj.assignment
    where position 210 = '512'
    (emp_id position 1 smallint,
     proj_id position 3 char(4),
     start_date position 13 date,
     end_date position 23 char(8) null if '01-01-01')

  into demoproj.consultant
    where position 210 = '222'

  into demoproj.expertise
    where position 210 = '320'
    (emp_id position 1 smallint,
     skill_id position 3 smallint,
     skill_level position 5 char(2) null if '99',
     exp_date position 7 date)

   into demoproj.project
     where position 210 = '416'

   into demoproj.skill
     where position 210 = '445';
```

Phased Load Procedure

Steps

Follow the steps shown next to perform a phased load:

Note: Optionally, back up the database areas between the load and build steps if you want to recover the data in the event of a failed job step.

Action	Statement
Identify the following tables: ■ All tables clustered through referential constraints; if multiple levels of clustering exist, the tables in each level must be loaded in a separate operation before those at a lower level ■ All referencing tables in linked index constraints where the referenced key is an index; if multiple levels of such a structure exist, the tables in the higher levels must be loaded before those at a lower level	
In local mode, load and build all tables not identified in Step 1 above.	LOAD with the NO VALIDATE option
■ For each clustering level,load and build all tables clustered through referential constraints ■ For each linked index level, load and build all tables that participate in linked index constraints	LOAD with the NO VALIDATE option
Validate the referential constraints of all the loaded tables	VALIDATE SEGMENT

Example

In this example, the tables BENEFITS, COVERAGE, EMPLOYEE, and POSITION are loaded in a phased load procedure. The tables have the following characteristics:

Table	Characteristics
BENEFITS	References EMPLOYEE in a linked, clustered constraint
COVERAGE	References EMPLOYEE in a linked, clustered constraint
EMPLOYEE	References DEPARTMENT in an unlinked constraint
POSITION	References EMPLOYEE in a linked, clustered constraint

To load the tables, load and build the EMPLOYEE table first, followed by the remaining tables. After all 4 tables are loaded, validate the referential constraints that exist between them. Each of these statements must be executed in a separate job step:

```
load
  into demoempl.employee
  where position 150 = '415'
  no validate;

load
  into demoempl.benefits
  where position 150 = '478'

  into demoempl.coverage
  where position 150 = '488'

  into demoempl.position
  where position 150 = '492'
  no validate;

  validate segment demoempl;
```

Segmented Load Procedure

Steps

Follow the steps listed next, to perform a segmented load:

Action	Statement
Load the input records in groups; for example, the first 1,000,000, the second 1,000,000 and so on	LOAD NO BUILD using the FROM and FOR clauses for each group of input records
Build the table indexes	BUILD INDEXES NO VALIDATE
Build the indexed constraints	BUILD CONSTRAINTS NO VALIDATE
Validate the referential constraints of the tables within the segment	VALIDATE

Example

This example uses a segmented load to load table EMPLOYEE, which contains more than 2,000,000 rows. By default, each input record is to be stored in the EMPLOYEE table, with each field in the input record corresponding in length and data type to each column defined for the EMPLOYEE table.

The first LOAD statement loads 1,000,000 rows in the table, starting with the first input record. CA IDMS/DB will notify the user for each 100,000 input records processed. The second LOAD statement processes the next 999,999 input records beginning with input record 1,000,001. The third LOAD statement processes the remaining input records.

Because the table is so large, indexes and indexed constraints are built in separate steps using the BUILD statements. Finally, the referential constraints for the table are validated.

```
load
  into demoempl.employee
  no build
  for 1000000
  notify 100000;

load
  into demoempl.employee
  no build
  from 1000001
  for   999999
  notify 100000;

load
  into demoempl.employee
  no build
  from 2000000
  notify 100000;

build indexes
  for demoempl.employee
  no validate
  notify 100000;

build constraints
  for demoempl.employee
  no validate
  notify 100000;

validate table demoempl.employee
  notify 100000;
```

Stepped Load Procedure

Steps

Follow the steps listed next, to perform a stepped load:

Note: If you want to be able to recover the database in the event of a failed job step, back up the database areas between each job step.

Action	Statement
1. In local mode, load one or more database tables choosing one of the following options. If you intend to build indexes and relationships for the tables immediately following the load step, choose one of the options that creates an intermediate work file:	
1.1Load, creating intermediate extract files for the build phase	LOAD STEP1 EXTRACT BOTH (the default)
1.2 Load, creating an intermediate extract file for building indexes	LOAD STEP1 EXTRACT INDEXES
1.3 Load, creating an intermediate extract file for building relationships	LOAD STEP1 EXTRACT RELATIONSHIPS
1.4 Load, creating no intermediate extract file	LOAD STEP1 NO EXTRACT
2. If you specified WITHOUT PRESORT, skip this step. Otherwise, sort the data using an external sort program and the sort cards supplied by CA IDMS/DB. Then continue the load phase of the stepped load procedure.	LOAD STEP2
3. If you specified LOAD STEP1 NO EXTRACT, perform this step to collect the data necessary to build the table indexes and indexed constraints	BUILD STEP1
4. Sort the data using an external sort program and the sort cards supplied by CA IDMS/DB	
5. After all of the tables have been loaded or after completingthe previous step, determine the db-keys of rows in any tables that participate as the referenced table in a linked index referential constraint	BUILD STEP2
6. Sort the data using an external sort program and the sort cards supplied by CA IDMS/DB	

Action	Statement
7. Build unclustered indexes and linked index referentialconstraints	BUILD STEP3
8. Sort the database using an external sort program and the sort cards supplied by CA IDMS/DB	
9. Update the prefixes of any tables that participate as thereferencing table in a linked index referential constraint	BUILD STEP4
10. Perform the first pass at validating the relationships between tables that participate in referential constraints	VALIDATE STEP1
11. Sort the database using an external sort program and the sort cards supplied by CA IDMS/DB	
12. Perform the second pass of validatingreferential constraints; generally, a second pass is required for unlinked relationships if either the referenced table or referencing table contains a CALC key.	VALIDATE STEP2

Example

In the next example, the DBA loads an SQL-defined database in the following steps:

- Loads tables CUSTOMER and INVENTORY using pre-sorted data
- Loads tables ORDERS and PARTS using pre-sorted data
- Using an area sweep, extracts information for building indexes and indexed constraints
- Builds the indexes and constraints for the table using separate steps and external sorts
- Validates referential constraints

```
load step1
  into custschm/customer
   where position 300 = '435'
  into custschm.inventory
   where position 300 = '457'
  without presort
  no extract;


load step1
  into custschm.orders
   where position 200 = '335'
  into custschm.parts
   where position 200 = '345'
  without presort
  no extract;


build step1
   for custschm.customer
       custschm.inventory
       custschm.orders
       custschm.parts;
```

Sort the data:

```
build step2;
```

Sort the data:

```
build step3;
```

Sort the data:

```
build step4;

validate step1;
```

Sort the data:

```
validate step2;
```

More Information

- For more information about the BACKUP, LOAD, BUILD, VALIDATE utility statements and the IDMSCALC utility program, see the *CA IDMS Utilities Guide*.
- For more information about SQL data types, see the *CA IDMS SQL Reference Guide*.
- For more information about loading a non-SQL defined database, see Chapter 22, “Loading a Non-SQL Defined Database”.
- For more information about DCMT commands, see the *CA IDMS System Tasks and Operator Commands Guide*.
- For more information about CA Culprit and CA OLQ, see the document set for each product.

Chapter 24: Monitoring and Tuning Database Performance

This section contains the following topics:

Monitoring Guidelines

Why You Need to Monitor

Eventually, a database may begin to outgrow its initial allocation of space, with resulting increased I/O and poor response time. If you don't monitor your databases on a regular basis, these conditions can become critical, forcing you to take emergency actions at an inconvenient time.

Suggested Monitoring Schedule

Consider using the following schedule as the basis for monitoring database performance:

Monitoring tool	Monitoring frequency	Information provided
JREPORT 004	Daily	Summary information on the database processing activities for each program recorded in the journal file
IDMSDBAN report 2	Weekly	Area detail statistics, such as number of logically full pages and number of relocated records
IDMSDBAN report 5	Monthly	Set detail statistics, such as the number of pages needed to store a chained set
PRINT SPACE	Daily	Area space utilization statistics
IDMSDBAN (all reports)	Monthly or as needed	Set statistics, including broken chains, record data, and area data

Keep a History of Meaningful Statistics

Keep a history of meaningful statistics so that you can identify abnormal conditions when they arise.

SQL Considerations

Most of the information in this chapter applies to both SQL and non-SQL defined databases. Text that applies to only one or the other will be noted. In addition, much of the chapter applies to the physical structures that underlie the database definition. Therefore, one set of terms will be used for these physical entities. For example, chain sets are the physical structure used to implement both SQL linked constraints and non-SQL sets defined with the MODE IS CHAIN clause.

Monitoring Facilities

Online and Batch Components

CA IDMS/DB offers the following online and batch tools for you to use to monitor the performance of your databases:

Facility	Uses
Performance Monitor	To monitor: ■ Real-time database and system statistics ■ System-wide, wait-time statistics for a unit of time Statistics about resource usage by individual programs
DCMT commands	To display definitions and run-time statistics for entities associated with a DC/UCF system
IDMSDBAN utility program	To check for broken chains and to display statistics and data for sets, records, and areas
OPER WATCH commands	To display dynamic run-time statistics associated with DC/UCF systems
PRINT INDEX utility statement	To monitor the structure of user-owned and system-owned indexes
PRINT SPACE utility statement	To monitor space utilization in segments or areas
PRINT JOURNAL utility statement	To display checkpoint information about transactions recorded on an archive or tape journal file
PRINT utility statement	To display the contents of requested database pages
JREPORTs	To monitor journal and database usage statistics

Facility	Uses
SREPORTs	To monitor system and database usage statistics
Online print log (PLOG)	To display system messages, system trace information, and snap dumps from the DDLDCLOG area
UPDATE STATISTICS utility statement	To refresh statistical information about SQL defined databases, and non-SQL defined databases that are accessed by SQL commands.

More Information

- For more information about Performance Monitor, see the *CA IDMS Performance Monitor User Guide*.
- For more information about utility programs and statements, see the *CA IDMS Utilities Guide*.
- For more information about DCMT and OPER commands, see the *CA IDMS System Tasks and Operator Commands Guide*.
- For more information about JREPORTs and SREPORTs, see the *CA IDMS Reports Guide*.

Database Statistics

What is Collected

The DBMS collects a number of statistics on a run-unit or SQL transaction level that are categorized as basic and extended statistics. The following tables provide a description of each.

Basic Statistics

Run-unit or SQL Transaction Level	Description
Pages read	Specifies the number of database pages physically read.
Pages written	Specifies the number of requests made to write a database page.
Pages requested	Specifies the number of times the DBMS requested that a database page can be read.
CALC record on target page	Specifies the number of times a CALC record was stored and the record occurrence fit on its target page.
CALC record overflow	Specifies the number of times a CALC record was stored and the record occurrence overflowed its target page.

Run-unit or SQL Transaction Level	Description
VIA record on target page	Specifies the number of times a VIA record was stored and the record occurrence fit on its target page.
VIA record overflow	Specifies the number of times a VIA record was stored and the record occurrence overflowed its target page.
Record requested	Specifies the total number of record occurrences the DBMS accessed to satisfy DML requests.
Records current of run-unit	Specifies the number of record occurrences that were made current of run-unit.
Calls to the DBMS	Specifies the total number of DML commands passed from the user application to the DBMS.
Fragments stored	Specifies the number of variable length record fragments (SR4) stored by the run-unit.
Relocated records	With a non-SQL defined database: Specifies the number of variable length record fragments or relocated records brought back to their original root or target page. With an SQL defined database: Specifies the number of records relocated from or returned to their original target page.
Locks requested	Specifies the total number of record locks requested.
Select locks held	Specifies the number of select (shared) locks held at the end of the run-unit.
Update locks held	Specifies the number of update (exclusive) locks held at the end of the run-unit.

Extended Statistics

Run-unit or SQL Transaction Level	Description
SR8 records split	Specifies the number of SR8 records that were split during the life of the run-unit.
SR8 spawns	Specifies the number of times that a new level of an index was created due to the splitting of the index's top level SR8.
SR8 records stored	Specifies the number of SR8 records of all levels that were stored into the database.
SR8 records erased	Specifies the number of SR8 records of all levels that were erased from the database.

Run-unit or SQL Transaction Level	Description
SR7 records stored	Specifies the number of SR7 records stored into the database.
SR7 records erased	Specifies the number of SR7 records erased from the database.
Total binary searches	Specifies the total number of times the DBMS initiated a binary search against an index.
Levels searched	Incremented every time that the DBMS goes down a level during a binary search throughout the life of the entire run-unit across all accessed indexes.
Orphans adopted	Specifies the number of orphaned user records that were adopted back to their referencing level-0 SR8.
Fewest levels searched	Specifies the fewest number of levels walked during a binary search throughout the life of the run-unit.
Most levels searched	Specifies the greatest number of levels walked during a binary search throughout the life of the run-unit.

Where to find the statistics

The database statistics are initially accumulated on the run-unit level and can be accessed through the following facilities:

- JREPORTS report on the basic statistics from information written to the journal files.
- ACCEPT 'statistics' DML commands allow the user to access both the basic and extended statistics from a currently running program.
- GET STATISTICS SQL DML commands allow an application program or the command facility to access both the basic and extended statistics for the current SQL transaction.

CA IDMS also consolidates the statistics from all run-units active for a CV task. These values can be accessed through the following facilities:

- SREPORTs report on both the basic and extended statistics from data written to the DCLOG.
- PMARPTs report on the basic statistics from data written by the ApplicationMonitor component of the Performance Monitor.

Items to Monitor and Tune

Monitoring the Database

For your database, the major areas of degradation are:

- Pages over 70% full
- CALC and VIA (clustered) record overflow
- Fragmented records
- Inefficient index structures
- An increase in logically-deleted or relocated records

Journal Use

Useful Statistics to Monitor

Statistic	Meaning	Action
Journal read waits	Indicates CA IDMS/DB must wait to read a page from a journal file into the journal buffer during a rollback operation.	Increase the number of pages in the journal buffer
Journal page utilization	Indicates the fullness of journal pages written from the journal buffer.	Create fuller journal buffers by: ■ Adjusting the journal buffer page size in the definition of the journal buffer ■ Increasing the journal TRANSACTION LEVEL option at system generation or using a DCMT VARY JOURNAL command

Where the Statistics are Reported

- ARCHIVE JOURNAL utility statement report
- JREPORT 004
- Performance Monitor
- DCMT DISPLAY JOURNAL

Buffer Utilization

Useful Statistics to Monitor

Statistic	Meaning	Possible action
Buffer utilization ratio	Indicates the ratio of the number of pages requested to the number read; values less than 2 indicate a problem with the buffer size or with the design of the database	■ Increase the number of buffer pages ■ Reassign files to buffers
Forced writes	Indicates the number of times CA IDMS/DB had to write a buffer page to storage in order to read a database page	■ Increase the number of buffer pages ■ Reassign files to buffers ■ Issue COMMITs more frequently in update jobs
Buffer waits	Indicates the number of times the buffer was requested but was not available	■ Increase the number of buffer pages ■ Reassign files to buffers

Where the Statistics are Reported

- Performance Monitor
- SREPORT 003
- DCMT DISPLAY/VARY BUFFER

Space Management and Database Design

Useful Statistics to Monitor

Statistic	Meaning	Possible Action
Clustering ratio	Indicates the ratio of the number of records requested to the number of pages requested; ratios less than 4 indicate poor database design or space availability problems	■ Redesign the database using clustering more effectively ■ Increase the area's page size or page range andunload and reload the database ■ Reassign files to buffers

Statistic	Meaning	Possible Action
Page space availability	Indicates how full database pages are	■ Increase the database page size ■ Increase the number of pages
Fragments stored	Indicates the number of fragments stored for a variable-length record.	■ Increase the page size and read each record in an update mode ■ Increase the page reserve size ■ Change fragmentation specifications
Records relocated	Indicates the number of expanded records moved to a new page due to lack of space	■ Unload/reload the database ■ Increase the page size and read each record in an update mode
CALC cluster ratio	Indicates the ratio of CALC records stored on the target page to the total number (that is, hits plus overflow) stored; values less than 1 indicate space availability problems	Increase the area's page size or number of pages and unload and reload the database
VIA cluster ratio	Indicates the ratio of VIA (or clustered) records stored on the target page to the total number (that is, hits plus overflow) stored; values less than 1 indicate large clusters, space availability problems, or small page size	Increase the area's page size or number of pages and unload and reload the database
Effectiveness ratio	Indicates the ratio of number of records CA IDMS/DB requests to the number that are current-of-run-unit. Values much higher than 1 indicate poor program logic or set options	Review application/database design. Consider use of PRIOR or OWNER pointers and possible elimination of some sorted sets. (Note that linked constraints in SQL-defined databases always include PRIOR and OWNER pointers.)
Logically deleted records	Indicates the number of logically deleted records	Physically delete the logically deleted records using the CLEANUP utility statement

Where Statistics are Reported

- JREPORT 004
- SREPORTs 003, 007, and 009
- Performance Monitor
- IDMSDBAN utility report 5
- PRINT SPACE utility statement report
- PRINT JOURNAL utility statement
- UPDATE STATISTICS utility statement report for the SQL catalog

Indexing Efficiency

Useful Statistics to Monitor

Statistic	Meaning	Possible action
Orphan count	Indicates the number of orphaned SR8 records.	Tune or rebuild the index if more than 25% of the member records are orphaned.
Index levels	Indicates the number of levels in the index.	Tune or rebuild the index if the number of levels exceeds the number originally calculated
SR8 split	Indicates the number of SR8 splits.	If the number of SR8 splits is high, determine if applications frequently insert a large group of index entries in one spot; tune or rebuild the index to balance it and cleanup orphan index records.

Where the Statistics are Reported

- Performance Monitor (Realtime monitor) Run Unit Detail screen
- PRINT INDEX utility statement
- IDMSDBAN utility report 5

Tuning an Index

Tuning of an index should be done when any of the following conditions exist:

- Index usage performance degradation
- Index needs redimensioning

One way to tune an index is to rebuild it: for a non-SQL defined database, use MAINTAIN INDEX, for an SQL-defined database, drop the index and recreate it.

An alternative way of index tuning is to use the TUNE INDEX utility statement. TUNE INDEX compares to rebuilding the index as follows:

	TUNE INDEX	**Rebuild Index**
Operating mode and area availability to applications	Local mode: offline CV and batch-CV: online	Local mode: offline
Specialized DMCL/schema needed	No	Yes
Attributes that can change	INDEX BLOCK CONTAINS and PAGE RESERVE	All

You can run the TUNE INDEX command to do the following:

- Enhance performance
  - Eliminate orphans at all levels of the index. This improves performance if the index is accessed at the bottom level.
  - Rebalance SR8's. In an unbalanced index, more records have to be accessed when traversing the index from the top to the bottom.
  - Resequence SR8's. Optimum performance is obtained if the top level SR8 resides on the same page as the index owner and if the bottom level SR8's are in physical sequence.
- Rebuild the index to accommodate future growth
  - Specify TEMPORARY PAGE RESERVE and TEMPORARY INDEX UTILIZATION to define the attributes with which to tune the index.

Run TUNE INDEX online or in batch through central version mode if the affected area or areas must remain online while the index is tuned.

Database Locks

Useful Statistics to Monitor

Statistic	Meaning	Possible action
Number of non-share locks held	Indicates the number of non-share locks (primarily update locks) held. The larger the number of update locks held, the greater the probability of contention between the tasks holding the locks and other tasks accessing the same database.	Issue COMMITs more frequently in update jobs
Task wait status	Indicates whether a task is waiting for access to an area or record	■ Tasks that are waiting on locks have an ECB type of 'LMGR Lock'.If you notice a task waiting a long time on one or more locks, review ready modes and database design, especially for contention for OOAK and FOAK records, by examining all tasks exhibiting this behavior for common programs, functions, and database references. ■ If overall throughput is constrained, identify the source; for example, CPU or DASD usage. ■ If overall throughput is not constrained, identify potential deficiencies in database or application design or implementation; for example, look at the number of locks held by individual programs; determine if tasks contend for OOAK and FOAK records in which case lowering the DEADLOCK DETECTION INTERVAL might improve the situation.

Statistic	Meaning	Possible action
ECB type	Denotes the type of resource being waited on. In the case of area locks and dbkey locks, this statistic will contain the literal 'LMGR ECB'. **Note**: in the Performance Monitor this information is listed under the column headings 'First ECB', 'Second ECB', and 'Third ECB'.	
Number of shared locks held	Indicates the number of share locks held. Share locks allow transactions other than the owning transaction to read the row, but not to update it. Thus, higher levels of share locks can impede concurrency (and throughput) if they are placed on rows in areas that are heavily accessed.	The number of locks held can be reduced by increasing the COMMIT frequency within the application.
ISO (SQL only)	Indicates the isolation level of the transaction. The isolation level of a transaction defines how long retrieval locks are held.	Ensure that the transaction is running in the appropriate isolation level for the level of data integrity required by the application.
State (SQL only)	Indicates the state of the transaction which defines how the transaction is affecting the data it is processing: ■ Read only (RO) specifies that the transaction is reading data but not adding or updating. ■ Read write (RW) specifies that the transaction intends to add and update data.	Ensure that the transaction state is appropriate for the type of processing being performed. Transactions that only read data should have a state of RO.

Statistic	Meaning	Possible action
Ratio of global resource lock requests to local lock requests	Indicates the number of times that CA IDMS had to acquire or alter a global lock on an area, page, or record in order to service the indicated number of local lock requests. The larger this ratio, the greater the inter-member contention for resources, since CA IDMS acquires global record and page locks only if contention exists between members.	■ Issue COMMITs more frequently in update transactions ■ Disperse frequently updated data across more pages within the area ■ Increase the size of the area, especially if frequently inserting or deleting data in an area that is more than 70 percent full ■ Split the workload between members to minimize inter-member contention for resources
Ratio of the number of global lock waits to the number of global lock requests.	Indicates the number of times that CA IDMS had to wait for a global lock request to complete. This ratio is a measure of one or more of the following types of contention: ■ Inter-member contention for transaction resources ■ False contention caused by synonyms when hashing to the global lock table ■ Contention for operating system resources such as channels	■ Use operating system tools to determine the nature of the contention ■ Take the actions outlined above to reduce inter-member contention for transaction resources ■ Increase the number of lock table entries to reduce false contention

Statistic	Meaning	Possible action
Number of times lock storage overflowed	Indicates the number of times that CA IDMS had to acquire lock storage dynamically in order to satisfy a lock request. The larger this number the more CPU cycles that were expended to satisfy lock requests. Additionally the storage pool may become fragmented since dynamically acquired storage may not always be releasable.	■ Examine the overflow details to determine the type of storage overflows that occurred ■ Determine the applicable base factor for the type of overflowing storage: ■ Session and class storage is based on the number of logical terminal elements (LTERMs) defined for the system. ■ Resource and proxy storage is based on the SYSLOCKS system definition parameter ■ XES Request storage is based on the maximum number of tasks specified in the system definition. ■ Increase the appropriate base factor (the number of LTERMs, SYSLOCKS, or maximum number of tasks) to increase the size of the initial storage allocation, and thus reduce the number of overflows.

Where the Statistics are Reported

- For area contention:
  - SREPORTs
  - JREPORT 006
  - Performance Monitor (Realtime monitor): Active User Task Detail, Active System Task Detail screen, Transaction Detail screen, and SQL Detail screen
  - DCMT DISPLAY ACTIVE TASKS
  - Area status codes from DCMT DISPLAY TRANSACTION transaction id
  - Area status codes from OPER WATCH DB
  - OPER WATCH TIME
- For record contention:
  - Status codes from OPER WATCH DB
  - Status codes from DCMT DISPLAY TRANSACTION transaction id
  - DCMT DISPLAY LOCK (shows longterm and notify locks held by logical terminals)
- For lock storage overflows:
  - DCMT DISPLAY LOCK STATS
- For inter-member contention in a data sharing environment:
  - DCMT DISPLAY LOCK STATS
  - DCMT DISPLAY DATA SHARING XES LOCKS

Reducing Area Contention

- Ready areas in shared ready modes
- Create a window for batch jobs

Reducing Record Contention

- Have the application issue more COMMITs
- Run applications that contend for a record serially, rather than concurrently
- Have some applications use a different access route that avoids the record under contention
- Change the database design so that access can be less serialized

Longterm Locks

Useful Statistics to Monitor

Statistic	Meaning	Possible action
Tasks having areas locked	Shows which tasks have areas locked	Use this information to identify tasks that ready an area in protected or exclusive mode, since this increases the potential for throughput degradation
Longterm/ notify locks	Displays longterm or notify lock statistics by logical terminal	Use this information to identity tasks that hold a large number of longterm and/or notify locks

Where the Statistics are Reported

- DCMT DISPLAY AREA
- DCMT DISPLAY LOCK AREA/LTERM

SQL Processing

Useful Statistics to Monitor

Statistic	Meaning	Possible Action
Sorts performed	The number of sorts performed as the result of an SQL statement (the result of processing the ORDER BY clause)	Add additional indexes or sorted constraints to reduce the number of sorts
Maximum rows sorted	The largest number of rows sorted as the result of an ORDER BY clause	Add additional indexes to eliminate the sort
AM recompiles	The number of times access modules were automatically recompiled at runtime because of a recompilation of the corresponding program or dialog, or because of a change in the underlying database definition.	Examine the cause of compilations. If necessary, move frequently altered tables to areas with table level stamp synchronization.

Where the Statistics are Reported

- SREPORTs
- Performance Monitor
- IDMSDBAN utility reports (database structure)

Reducing I/O

I/O can be reduced through:

- Caching files in memory
- Database reorganization
- Application design
- Database design
- The UPDATE STATISTICS utility command (for SQL-accessed databases)

Each of these is discussed as follows.

By Caching Files in Memory

If a database file is cached in memory, the DBMS looks in the cache before reading a database page from disk. If the page resides in the cache, the retrieval I/O is eliminated. If the page must be read from the disk, it is saved in the cache to satisfy future retrieval operations. Database files with a high number of I/Os are good candidates for caching in memory.

There are two basic types of file caching: shared cache which uses coupling facility services to enable a single cache to be shared by multiple central versions and memory cache which is accessible only by a single central version. The remainder of this discussion focuses on memory cache. For information about shared cache, see the *CA IDMS System Operations Guide*.

Note: Memory caching is available only for non-native VSAM files.

To enable the use of memory cache, take the following steps:

- Decide which files to cache by using standard performance-monitoring tools to determine the database files with the most I/O. For example, you can use the DCMT DISPLAY STATISTICS FILES to get a list of all files and their associated I/O counts or look at gathered operating system statistics. Choose files with the highest retrieval counts.
- Change the DMCL definition to specify MEMORY CACHE YES for each file to be cached. For details, see DMCL Statements. Alternatively, use the DCMT VARY FILE command to dynamically initiate the use of memory cache for one or more files. For more information, see the *CA IDMS System Tasks and Operator Commands Guide*.
- Compute the total amount of storage that is needed to cache the selected files. To do this, for each file, multiply the number of blocks in the file by the file's block size and total the results. The resulting value is the amount of storage needed. Ensure that sufficient storage of the required type is available to all jobs that use the altered DMCL.

If the operating system supports 64-bit storage, the cache is allocated in 64-bit storage if sufficient memory is available. If no or not enough 64-bit storage is available to hold the entire file, the file will not be cached in memory. For details, see 7.13, "DMCL Statements".

Note: For more information about operating-specific considerations in using memory cache and 64-bit storage, see the *CA IDMS System Operations Guide*.

Through Database Reorganization

Database reorganization includes:

- Reducing full pages by changing the size of a database page or increasing the number of pages
- Reducing overflow by changing the size of a database page or increasing the number of pages
- Decreasing fragmentation for non-SQL defined databases by:
  - Specifying page reserve
  - Changing page size
  - Reassigning records
  - Redefining fragmentation specifications
  - Increasing the number of pages
- Increasing the efficiency of an index's structure by decreasing the number of levels in the index and/or assigning SR8 records to a separate page range
- Reducing logically deleted and/or relocated records by physically deleting logically deleted occurrences using the CLEANUP utility statement and/or unloading and reloading the data
- Reducing the number of fragments and/or relocated records by increasing the page size and reading all records in an update mode

More Information

- For more information about changing page size, see Chapter 27, “Modifying Physical Database Definitions"
- For more information about modifying indexes, see Chapter 31, “Modifying Indexes, CALC Keys, and Referential Constraints” and Chapter 33, “Modifying Schema Entities”
- For more information about reassigning records and redefining fragmentation specifications, see Chapter 33, "Modifying Schema Entities".
- For more information about utility statements, see the *CA IDMS Utilities Guide*.

Through Application Design

Selecting the Optimal Path

The first step to determine if the application is optimally designed is to determine if it is accessing the data it needs, using the access path that will create the fewest number of I/Os. To determine if this is true:

1. Walk through the application and identify the actual transaction path
2. Review the existing database design and determine if there is a more efficient way to:
   - Access the needed records
   - Process the necessary relationships

Through Database Design

Take into account the following database design considerations for reducing I/O:

- Adding sets, indexes, pointers, redundancy
- Changing set type, set (index) order for non-SQL defined databases
- Changing location (area) of record or index, index and/or set stored VIA (or clustered)
- Changing UNLINKED constraints to LINKED (SQL-defined databases) or repeating item (non-SQL defined databases)
- Splitting a record

By Using UPDATE STATISTICS (SQL-Accessed Databases)

When to Use UPDATE STATISTICS

Execute the UPDATE STATISTICS utility statement at the following times:

- Periodically (according to the needs of the application) to reflect shifts in the distribution of data in the database (for example, changes in owner/member ratios, area space utilization, index layout)
- After individual applications that alter the distribution of data; for example, monthly or year-end summary and offload processing

Use UPDATE STATISTICS on SQL-Defined Tables or Areas

Run UPDATE STATISTICS on individual tables or whole areas. The resulting statistics are stored in the SQL catalog and are used by the Access Module Compiler to generate optimal access strategies for SQL processing. Access modules that reference the tables whose statistics have been updated can then be recompiled to take advantage of the updated information. Table/access module cross-reference information on the catalog can be used to determine which access modules to recompile.

Use UPDATE STATISTICS on NON-SQL Schemas If They are Accessed by SQL

Run UPDATE STATISTICS on some or all areas defined in a non-SQL Schema. The resulting statistics are kept in the dictionary that defines the non-SQL schema. If the database is accessed by SQL the statistics will be used by the Access Module Compiler to generate optimal access strategies for SQL processing.

Restrictions on Statistics and Non-SQL Schemas

Non-SQL statistics are kept with the schema definition in the dictionary. This means statistics may be kept for only one physical database per schema. When processing an SQL command, only the current set of statistics will be used for that command regardless of the physical database being accessed by that command. The user must decide which physical database will provide the statistics that best meets their needs and run UPDATE STATISTICS against that database.

Chapter 25: Dictionaries and Runtime Environments

This section contains the following topics:

Dictionaries

A dictionary is a special CA IDMS database that contains definitions of other databases, DC/UCF systems, and applications. Information in the dictionary is organized into *entity types* that correspond to major data processing components (for example, tables, records, programs). The dictionary becomes populated with information about the data processing environment as various CA IDMS/DB software components are executed.

System and Application Dictionaries

Each DC/UCF system must contain a system dictionary. Any number of application dictionaries can also exist in a CA IDMS/DB runtime environment. The following table describes both types of dictionaries:

Dictionary	Description
System	Includes all information required to establish, maintain, and control the processing environment: ■ The DC/UCF system definition ■ The physical database definitions Each runtime environment must have a system dictionary named SYSTEM.
Application	Optional dictionaries that contain information specific to a particular application, group of applications, or development groups: ■ The logical database definitions ■ Maps, dialogs, records, programs, elements A runtime environment may contain zero or more application dictionaries the names of which are user-defined.

Physical Components of a Dictionary

Dictionary Areas

Dictionaries are composed of the following areas:

Area name	Description
DDLDML	Contains the following types of information: ■ DC/UCF system definitions ■ Non-SQL schema and subschema definitions ■ Maps ■ Dialogs ■ Source modules ■ Record and element descriptions ■ IDD users ■ Classes and attributes
DDLDCLOD	Contains load modules associated with entities contained in the DDLDML area; for example: ■ Map load modules ■ Dialog load modules ■ Subschema load modules
DDLCAT	Contains definitions of physical databases (segments, DMCLs, database name tables); at sites with the SQL option, contains definitions of SQL entities (tables, constraints, indexes, and so on)
DDLCATX	Contains indexes defined on entities stored in the DDLCAT area
DDLCATLOD	Contains: ■ DMCL load modules ■ Database name table load modules ■ Access modules at sites with the SQL option
DDLDCMSG	Contains system and user-defined messages

Logical Components of a Dictionary

Dictionary Components

You can group the six areas of the dictionary into logical components based on the inherent relationships that exist between the dictionary areas:

Logical Component	Dictionary Areas
Base definition component	DDLDML DDLDCLOD
Message component	DDLDCMSG
Catalog component	DDLCAT DDLCATX DDLCATLOD

Components of a System Dictionary

A system dictionary always contains all three components:

- A base definition component
- A catalog component
- A message component

Components of an Application Dictionary

An application dictionary may contain all or a subset of the components. At sites without the SQL option, an application dictionary usually contains only a base definition component and a shared message component.

Sharing the Message Area

In most cases, an application dictionary will not have its own message area. Since the runtime system uses only the system message area (SYSMSG.DDLDCMSG) to display messages, most application dictionaries will share the system message area, rather than having a separate message area.

Assigning Dictionary Areas to Segments

Segment by Component

The six areas that make up a dictionary should be segmented by logical component. That is, a segment should be defined for each of the base definition, catalog, and message components of a dictionary.

In most cases, a dictionary will not have its own message component, but will share the system message area SYSMSG.DDLDCMSG. Sites without the SQL option do not need to define a catalog segment for their application dictionary.

Define a Database Name

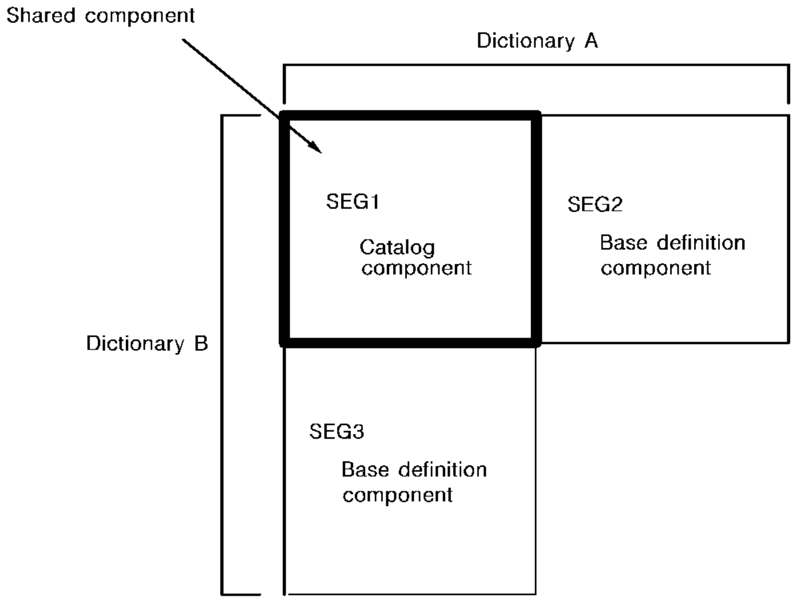
If a dictionary is made up of more than one segment, you must define a database name to represent the dictionary. The database name identifies all of the segments that together make up the dictionary.

The one exception to this is a dictionary comprised of only two segments, one of which is the SYSMSG segment. A database name is unnecessary because CA IDMS/DB automatically uses the system message area (in the SYSMSG segment) if no message area is associated with the dictionary.

Sharing Dictionary Areas

Sharing Components

By separating dictionary components into segments, you can share those components between dictionaries, as illustrated next:



To share SEG1 between dictionary A and dictionary B, define a database name for each that includes the SEG1 segment.

System Dictionary Components

You should not share the base definition component and the catalog component of the system dictionary with application dictionaries. Since the system dictionary contains critical information needed to control and execute your CA IDMS environment, it should be accessed only by authorized personnel and should be reserved for the following information:

- DC/UCF system definitions
- Physical database definitions

Sharing Individual Areas

It is possible to separate a component into multiple segments so that individual areas (such as a load area) can be shared across dictionaries. While this is supported, it is not recommended because of the potential for naming conflicts between the dictionaries. For example, a dialog in one dictionary could have the same name as a map in another dictionary, both of which have an associated load module.

Important: Under no circumstances should the DDLCAT and DDLCATX areas be placed in different segments.

Page Groups

All segments associated with a dictionary must have the same page group (and maximum number of records per page). If you have different page groups, you will receive errors when you attempt to access the dictionary through IDD or other dictionary tools.

This rule also applies to the system message area (SYSMSG.DDLDCMSG). It can only be included in dictionaries whose other segments have the same page group as the SYSMSG segment. When processing a dictionary with a difference page group, IDD cannot be used to display or update messages. Maintenance of the system message area can only be done from a dictionary that has the same page group as the SYSMSG segment.

Page Groups and SQL

When defining an application dictionary that contains a catalog component, the page groups of the base and catalog components may be different. The page group of the catalog component has no impact on the page group of data that may be accessed while connected to the dictionary.

CA-supplied Dictionary Definitions

Provided on Install Tape

As part of installation, you receive definitions for entities required to operate your CA IDMS environment. These definitions are described next:

Definition	Description
Non-SQL descriptions of the dictionary	A schema and subschemas describing the base definition and message components and that part of the catalog component used for physical database definitions

Definition	Description
At sites with the SQL option, an SQL description of the catalog component	Table definitions of the catalog component of the SYSTEM schema and views based on those tables in the SYSCA schema
Runtime messages	Messages used by CA-written software
Entity, class, and attribute definitions	Definitions of base entity, class, and attributes used by CA IDMS tools
Protocols and standard error routines	Generalized source modules that the DML processors use to convert DML statements into calls for DBMS services
DC/UCF device types, task, and program definitions	Definitions used to generate DC/UCF systems
CA Culprit report modules	CA Culprit source modules used to produce standard reports; for example, JREPORTs, SREPORTs, and DREPORTs
Nondatabase structures	Records that are not associated with a CA IDMS database. CA IDMS/DB stores the definitions of nondatabase structures as records in the dictionary; applications can copy the definitions of the records at compile time by means of COPY IDMS or INCLUDE IDMS compiler-directive commands.

How the Dictionary Gets Populated

Dictionaries are populated with CA-supplied definitions in one of three ways:

Definition	Description
IDMSDIRL	Loads the non-SQL schema and subschemas that define the base definition and message components of the dictionary
IDD, IDMSCHEM, IDMSUBSC	Populates the base definition and message components of the dictionary using source members provided at installation
Command facility	Populates the catalog component of the dictionary with system table and view definitions (SQL-option only)

Where Information Should Reside

The information listed above can reside in either a system dictionary or an application dictionary, or both:

Information	Where it should reside
Non-SQL schema and subschema describing the dictionary	In *one* dictionary associated with each system; the definitions may be shared across systems
SQL definitions	In all dictionaries having a catalog component (SQL-option only)
Messages	In all system message areas
Entity, class, and attribute definitions	In all system and application dictionaries
Protocols and standard error routines	In all application dictionaries
DC/UCF device types, task, and program definitions	In all system dictionaries
CA Culprit report modules	In the same dictionary that contains the non-SQL schema and subschema definitions of the dictionary

Logical Database Definitions

CA-Supplied Schema

The following table describes the non-SQL schema supplied by CA that describes a dictionary. Its definitions are stored in a dictionary by the IDMSDIRL utility.

Schema	Areas
IDMSNTWK	DDLDML DDLDCLOD DDLCAT DDLCATX DDLDCMSG

CA-Supplied Subschemas

The following table describes subschemas supplied by CA and the CA IDMS products or facilities that make use of them. Most of these subschemas are distributed as object modules only. The source definitions of IDMSNWKA and IDMSNWKG are also stored in a dictionary by IDMSDIRL for user-reporting purposes.

Subschema	Areas	Used By
IDMSCATL	DDLCATLOD	■ Loader processing ■ CLOD DC/UCF system task ■ PUNCH utility statement Database administrators when executing utilities such as UNLOAD/RELOAD against the DDLCATLOD area
IDMSCATZ	DDLCAT DDLCATX DDLCATLOD	■ The command facility for SQL processing andphysical database definition ■ User applications issuing dynamic SQL requiring automatic recompilation of an access module or issuing SQL DDL statements ■ Database administrators when executing utilities such as UNLOAD/RELOAD against the DDLCAT and DDLCATX areas
IDMSNWKA	DDLDML DDLDCLOD DDLDCMSG	■ IDD DDDL compiler (IDMSDDDL) ■ DC/UCF system generation compiler (RHDCSGEN) ■ DC/UCF startup ■ Schema and subschema compilers (IDMSCHEM and IDMSUBSC) ■ CA IDMS/DC mapping compilers (MAPC and batch) ■ CA ADS compilers ■ CA OLQ ■ CA Culprit ■ The Automatic System Facility (ASF)
IDMSNWKL	DDLDCLOD	Loader processing and the CLOD DC/UCF system task

Subschema	Areas	Used By
IDMSNWKT	DDLDML	SQL processing to access non-SQL defined database descriptions
IDMSNWKU	DDLDML DDLDCLOD DDLDCMSG DDLCAT DDLCATX	Database administrators when executing utilities such as UNLOAD/RELOAD against dictionary areas DDLDML, DDLDCLOD, and DDLDCMSG
IDMSNWKG	DDLDML DDLDCLOD DDLDCMSG DDLCAT DDLCATX	IDMSRPTS
IDMSNWK6	DDLDCMSG	System message processing
IDMSNWK7	DDLDCRUN	QUED and QUEM DC/UCF system tasks and queue processing
IDMSNWK8	DDLDML	CLIST and send-message processing
IDMSNWK9	DDLDCLOG	Online print log (OLP) and PRINT and ARCHIVE LOG utility statements

Note: Additional non-SQL schemas and subschemas are supplied at installation time. For more information, see the *CA IDMS Security Administration Guide*.

SQL Table Definitions

At sites with the SQL option, CA IDMS/DB also provides the table and view definitions that describe the catalog component of the dictionary. These definitions are distributed under two schema names:

Definition	Description
SYSTEM	Contains the catalog table definitions; no changes can be made to any of the entities in the SYSTEM schema
SYSCA	Contains the CA-supplied views of the SYSTEM tables and records in the IDMSNWK schema; these views restrict access to table definition information based on a user's SELECT authority on the table.

Note: For more information and a description of the table definitions, see the *CA ADS Reference Guide*.

Protocols, Nondatabase Structures, and Modules

The following table summarizes the protocols, nondatabase structures, and modules placed in the dictionary at installation time:

Language	Protocol	Non-database structure	Module
COBOL	BATCH	DB-STATISTICS	IDMS-STATUS for
	BATCH-AUTOSTATUS	SUBSCHEMA-CTRL for	BATCH-AUTOSTATUS
	CICS	IDMS-DC	IDMS-DC
	CICS-AUTOSTATUS	IDMS-DC-NONAUTO	DC-BATCH
	CICS-EXEC	DC-BATCH	all others
	CICS-EXEC-AUTO	CICS	
	CICS-STANDARD	CICS-AUTOSTATUS	
	CICS-STD-AUTO	CICS-EXEC	
	DC-BATCH	CICS-EXEC-AUTO	
	IDMS-DC	CICS-STANDARD	
	IDMS-DC-NONAUTO	CICS-STD-AUTO	
	IDMSDML-PROTOCOL-SQL	SUBSCHEMA-LR-CTRL	
PL/I	BATCH	DB-STATISTICS	IDMS_STATUS
	CICS	SUBSCHEMA_CTRL for	IDMS_STATUS
	CICS_EXEC	CICS	(IDMS_DC)
	DC_BATCH	CICS_EXEC	
	IDMS_DC	IDMS_DC	
	IDMSDML_PROTOCOL_SQL	DC_BATCH	
		SUBSCHEMA_LR_CTRL	
Assembler	BATCH	SSCTRL for	DBSTATS
	CICS	CICS	
	CICS-AUTOSTATUS	CICS-AUTOSTATUS	
	CICS-EXEC	CICS-EXEC	
	CICS-EXEC-AUTO	CICS-EXEC-AUTO	
	IDMSDC	SSLRCTL	

Defining New Dictionaries

Defining New Catalog Components

Physical Characteristics

The segment definition for all catalog components must have the following characteristics:

- The names of the areas must be DDLCAT, DDLCATX, and DDLCATLOD
- The page size of the areas should be at least 4856 plus page reserve

All other physical characteristics can be chosen based on processing requirements, hardware configuration, and standard database design techniques. For example, choose an access method and page size appropriate for your disk devices and consider using a page reserve on the DDLCATX area.

Catalog Components for Non-SQL Use

Without the SQL option, only a system dictionary requires a catalog component. When defining the corresponding segment, specify FOR NONSQL (or take the default).

Catalog Components For SQL Use

If the SQL option has been installed at your site, one or more of your application dictionaries will have an associated catalog component in order to define tables and views. The corresponding segment must have the following attributes:

- FOR SQL specification on the segment
- STAMP BY AREA for the DDLCAT and DDLCATLOD areas
- STAMP BY TABLE for the DDLCATX area

The catalog associated with the system dictionary can also be defined with these attributes. If it is, SQL can be used to examine the physical database definitions stored in the system dictionary.

When a new SQL catalog component is defined, take the following steps after the new segment has been formatted:

1. Define the system tables and views in the new catalog using the TABLEDDL and VIEWDDL members in the installation source library
2. Issue the UPDATE STATISTICS utility statement against the new DDLCAT area
3. Grant appropriate authorities to permit authorized users to create schemas in the new dictionary

Defining New Application Dictionaries

Steps

To create a new application dictionary, follow these steps:

Action	Steps
Start a session in the command facility	CONNECT TO SYSTEM
Define segments for the base definition and the catalog components of the dictionary Note that you need the catalog component only if the SQL option is installed at your site.	CREATE SEGMENT
Add the new segment(s) to the DMCL used at runtime	ALTER DMCL with the ADD SEGMENT clause
If you created two segments, define a new database name in the database name table	CREATE DBNAME
Generate, punch, and linkedit the new DMCL	See Chapter 27, "Modifying Physical Database Definitions"
If you created a new database name, generate, punch, and linkedit the new database name table	See Chapter 28, "Modifying Database Name Tables"
Create and format new dictionary files	See Chapter 17, "Allocating and Formatting Files"
Make the DMCL available to the runtime system	See Chapter 27, "Modifying Physical Database Definitions"
Populate the dictionary with CA-supplied definitions	Use IDD DDDL statements to add entity, class, and attribute definitions, protocols, and standard error routines
If you created a new catalog component: ■ Populate it with the system table and view definitions ■ Execute UPDATE STATISTICS for the DDLCAT area of the new dictionary ■ Grant appropriate authorities to define schemas in the new dictionary	

Example

The following example illustrates how to define a new application dictionary. It consists of a new definition component in segment TESTDICT, a new catalog component in segment TESTCAT, and the system message component.

The database name for the dictionary is TESTDICT.

1. Define the new TESTDICT and TESTCAT segments and their areas and files.

```
create segment testdict
  for nonsql
  page group 0
  maximum records per page 255;

add file testdml
  assign to testdml
  dsname 'test.ddldml';

add file testlod
  assign to testlod
  dsname 'test.ddldclod';

add area ddldml
  primary space 10000 pages
    from page 5000001
  maximum space 20000 pages
  page size 4276
  within file testdml
    from 1 for all blocks;

add area ddldclod
  primary space 1000 pages
    from page 5020001
  maximum space 5000
  page size 8196
  within file testlod
    from 1 for all blocks;
```

```
create segment testcat
  for sql
  page group 0
  maximum records per page 255
  stamp by area;

add file testcat
  assign to testcat
  dsname 'test.testcat';

add file testcatx
  assign to testcats
  dsname 'test.testcatx';

add file testcatl
  assign to testcatl
  dsname 'test.testcatl';
add area ddlcat
  primary space 5000 pages
    from page 5030001
  maximum space 10000 pages
  page size 8196
  within file testcat;

add area ddlcatx
  primary space 1000 pages
    from page 5040001
  maximum space 3000 pages
  page size 8196
  within file testcatx;

add area ddlcatlod
  primary space 500 pages
    from page 5045001
  maximum space 5000 pages
  page size 8196
  within file testcatl;
```

2. Modify the DMCL

3. Generate, punch, and linkedit the new DMCL:

   ```
   generate dmcl idmsdmcl;
   ```

4. Define a new database name for the dictionary

   ```
   add dbname alldbs.testdict
     segment testdict
     segment testcat
     segment sysmsg;
   ```

5. Generate, punch, and link the database name table:

   ```
   generate table dbtable alldbs;
   ```

6. Create and format new dictionary files:

   ```
   format segment testdict;
   format segment testcat;
   ```

7. Populate the dictionary using the appropriate source from the installation source library.

8. Execute UPDATE STATISTICS for the new DDLCAT area:

   ```
   update statistics for area testcat.ddlcat;
   ```

9. Assign appropriate authorities within the new dictionary.

Defining New System Dictionaries

Steps

To create a system dictionary for a new system, follow these steps:

Action	Steps
Start a session in the command facility	CONNECT TO SYSTEM

Action	Steps
Create new segments that contain these dictionary areas ■ DDLDML ■ DDLDCLOD ■ DDLCAT ■ DDLCATX ■ DDLCATLOD ■ DDLDCMSG **Note:** Use segment names that are different than existing segment names.	CREATE SEGMENT
If you created more than one segment, create a database name table entry that contains all the segments you created	CREATE DBNAME
Add the segment(s) to the DMCL	ALTER DMCL with the ADD SEGMENT clause
Generate, punch, and link the DMCL	See Chapter 27, "Modifying Physical Database Definitions"
If you created more than one segment, generate, punch, and link the database name table	See Chapter 28, "Modifying Database Name Tables"
Format the new dictionary files	FORMAT FILE/SEGMENT
Grant appropriate administrative privileges to authorized individuals on and within the new dictionary	See the *CA IDMS Security Administration Guide*
Re-define the dictionary segment(s) in the new dictionary by either: ■ Creating the segments directly ■ Punching the segment definitions from the current SYSTEM dictionary and re-adding them to the new dictionary Make sure the segment name of the message area in the new dictionary is SYSMSG. Define additional segments necessary for a complete runtime environment.	■ CREATE SEGMENT ■ PUNCH SEGMENT

Action	Steps
Define a database name table that includes the database name SYSTEM; SYSTEM must identify the new dictionary segments. Add additional entries as necessary.	■ CREATE DBTABLE ■ CREATE DBNAME
Create a new DMCL with associated database buffers,a journal buffer, and journal files	See Chapter 5, "Defining, Generating, and Punching a DMCL"
Add the new segments and associate the database name table with the new DMCL	ALTER DMCL
Generate, punch, and link the new DMCL	See Chapter 5, "Defining, Generating, and Punching a DMCL"
Generate, punch and link the new database name table	See Chapter 6, "Defining a Database Name Table"
Populate the system dictionary with the following CA-supplied definitions: ■ Entity, class, and attribute definitions ■ DC/UCF device types, tasks, and programs	
If the new catalog segment was defined as FOR SQL, complete its definition.	See 25.3.1, "Defining New Catalog Components".

Establishing a Default Dictionary

What is a Default Dictionary

A default dictionary is the dictionary that will be accessed by CA IDMS tools if you don't specify a dictionary by other means such as using a DCUF SET DICTNAME command or a CONNECT statement.

Defining a Default Dictionary

To define a default dictionary for your runtime environment, include a subschema mapping in the database name table associated with the runtime DMCL for the IDMSNWK subschemas. For example, the following statement establishes TESTDICT as the default dictionary for the runtime environment using the ALLDBS database name table:

```
create dbtable alldbs
   add subschema idmsnwk? maps to idmsnwk? dbname testdict;
```

Runtime Environments

Central Version or Local Mode

CA IDMS/DB can run within a DC/UCF system as a central version or in local mode:

- Central version operations provide database services to batch or online applications. Multiple users can gain access to a database concurrently.
- Local mode operations are batch operations that do *not* run under a central version. In local mode, only one user at a time has access to a database area in update mode.

Data Sharing Environment

Data sharing is an environment in which two or more central versions operate cooperatively through the use of a coupling facility. In this environment, multiple central versions may concurrently access a database area in update mode.

Central Version Runtime Components

The following table lists the components needed for a central version runtime environment:

Component	Description
System dictionary	Defines the DC/UCF system and physical database entities
DDLDCLOG	Contains central version log records when the log file for the central version is assigned to the database
DDLDCRUN	Contains runtime queue information used by CA-supplied tools and online user programs
DDLDCSCR	Contains runtime scratch information used by CA-supplied tools and online user programs
SYSMSG.DDLDCMSG	Contains CA-supplied and user-defined messages
DDLSEC	Contains user and group information
Application dictionaries	
User databases	
SYSTRK reference	Contains a description of the central version's database environment.

Considerations

- The segment name of the system message area must be SYSMSG.
- The segment(s) associated with DDLDCLOG and DDLDCRUN must be included in the SYSTEM database name.
- Each central version must have its own DDLDCLOG. In a non-data sharing environment, each central version must also have its own DDLDCRUN area. In a data sharing environment, the DDLDCRUN area may be shared among members of a data sharing group.
- The DDLDCSCR component is not needed if scratch information is maintained in memory. If the DDLDCSCR component is used, each central version must have its own and the segment associated with the DDLDCSCR must be included in the SYSTEM database name.
- The DDLSEC area may not be necessary depending on your security implementation.
- The SYSTRK reference is needed if change tracking is in effect for the central version.

More Information

- For more information about sharing the DDLDCRUN area, see the *CA IDMS System Operations Guide*.
- For more information about security, see the *CA IDMS Security Administration Guide*.
- For more information about specifying the location of scratch information, see the *CA IDMS System Generation Guide*.
- For more information about Change Tracking and referencing SYSTRK files, see "Change Tracking" in the *CA IDMS System Operations Guide*.

Local Mode Runtime Components

The following table lists the components needed for a local mode runtime environment:

Component	Description
System dictionary	Defines the DC/UCF system and physical database entities
SYSMSG.DDLDCMSG	Contains CA-supplied and user-defined messages
DDLSEC	Contains user and group information
DDLOCSCR	Contains runtime scratch information used by local mode CA-supplied tools and user programs issuing SQL requests
Application dictionaries	

Component	Description
User databases	
SYSTRK reference	Enables the local mode application to share the database environment of a CV.

Considerations

The segment name of the system message area must be SYSMSG.

- The DDLSEC area may not be necessary depending on your security implementation.
- The DDLOCSCR is always optional. If it is not available, scratch information is stored in memory or in the DDLDCSCR area.
- At least the default dictionary should be available in local mode. Additional application dictionaries may be needed for loading subschemas and processing SQL requests.
- Including a reference to a SYSTRK file is only available if the central version is using change tracking.

More Information

- For more information about security, see the *CA IDMS Security Administration Guide*.
- For more information about specifying the location of scratch information, see the *CA IDMS System Generation Guide*.
- For more information about Change Tracking and referencing SYSTRK files, see "Change Tracking" in the *CA IDMS System Operations Guide*.

SYSIDMS Parameter File

About SYSIDMS Parameters

A SYSIDMS parameter is a parameter that can be added to the JCL stream of a batch job running in local mode or under the central version. You can use SYSIDMS parameters to specify:

- Physical requirements of the environment, such as the DMCL and database to use at runtime
- Runtime directives that assist in application execution
- Operating system-dependent file information

For a complete list of the parameters that can be specified, see the *CA IDMS Common Facilities Guide*.

Establishing Session Options

Established at Signon

CA IDMS establishes options for your runtime session when you signon on to a DC/UCF system or when CA IDMS/DB issues its first database request from a batch application (in local mode or under the central version) or external teleprocessing monitor. The manner in which CA IDMS implements the options and how they affect your session depends on the runtime environment.

Specifying a Default Database or Dictionary

CA IDMS/DB provides several ways to specify a session default dictionary or database. The methods available depend on the runtime environment.

Online Processing

To specify a session default in an online environment, you can:

- Specify DICTNAME/DICTNODE or DBNAME/DBNODE attributes in a system or user profile
- Issue a DCUF command
- Issue a compiler SIGNON or CONNECT statement naming the dictionary and/or database from within an online CA IDMS/DB compiler or tool (this will update the default dictionary for the runtime session)

Batch Processing

To specify a session default dictionary for a batch central version or external teleprocessing monitor application, you can use:

- An IDMSOPTI module (for non-SQL applications only)
- A SYSCTL file
- A SYSIDMS parameter file

Note: For more information about how CA IDMS/DB determines which database or dictionary to access when provided with information by the program, IDMSOPTI module, SYSCTL file, and SYSIDMS file, see the *CA IDMS System Operations Guide*.

Local Mode Processing

To specify a session default for local mode, you can use:

- An IDMSOPTI module (for non-SQL applications only)
- A SYSIDMS parameter file

More Information

- For more information about database name tables, see Chapter 6, “Defining a Database Name Table".
- For more information about the SYSCTL file and IDMSOPTI module, see the *CA IDMS System Operations Guide*.
- For more information about dictionary entities, see the *CA IDMS IDD DDDL Reference Guide*.
- For more information about SYSIDMS parameter syntax, see the *CA IDMS Common Facilities Guide*.

Chapter 26: Migrating from Test to Production

This section contains the following topics:

Migration

Migrate Definitions from One Dictionary to Another

Whether you have multiple dictionaries under a single CA IDMS/DB system or several dictionaries under separate CA IDMS/DB systems, you probably need to migrate definitions from one dictionary to another. Typically, migration occurs when testing is complete and an application is ready for production. At that time, the database and application definitions must be moved from the test into the production environment.

Considerations for Non-SQL and SQL Defined Data

The need to migrate applies to SQL-defined and non-SQL defined databases and to applications using SQL or navigational DML. Most of this chapter applies to both SQL and non-SQL equally. Text that applies specifically to one or the other will be noted.

Migration Aids

CA IDMS Dictionary Migrator is a product supplied by CA that can be extremely helpful for migrating applications from one environment to another. For more information, see the *CA IDMS Dictionary Migrator User Guide*.

Establishing Migration Procedures

Considerations

Because many of the pieces of an application, such as subschemas, maps, and dialogs, exist in both source and load module format, you must consider the following questions when you migrate from one dictionary to another:

- Should you copy or move the components?
- Should you migrate and recompile source code to produce load modules?
- Should you migrate just the load modules?

Accessibility of the Source Code

The major benefit of a complete, fully documented application is that the proper source code is accessible when needed for debugging. If a problem arises and the source code resides in a properly controlled production environment, the source code can easily be found and it will correspond exactly to the load module(s) where the problem was encountered.

Availability of Disk Space

A trade-off to migrating a fully documented application is the amount of disk space required. The space may be in one environment, such as production, or may be spread out over a number of environments, such as development, test, and production. Determining exactly how much disk space is necessary depends on whether you decide to copy the application into the production environment or simply move it.

Redundancy

If you choose to maintain separate copies of the application, you must contend with the trade-offs of redundancy. Often, updates to one copy must also be made to the other, and they both must be made within a short period of time to maintain consistency.

Accessibility of Information

If you maintain only one copy of the application, you use a minimum amount of disk space and do not have to contend with redundancy. However, accessibility of information becomes a consideration. If the information is secured so that only one person is able to access it, procedures must be developed that allow maintenance programmers and all members of the staff to obtain reports of component definitions. At the same time, you must ensure that there is ample security so that no one can make accidental or malicious updates that would invalidate production applications.

Implementing Migration Procedures

Steps

There are essentially four steps involved in migration:

1. Determine the types of components to migrate

   Carefully examine the circumstances for dependencies and other relationships among the components involved.

2. Determine the sequence of migration

   Components that do not depend on the definitions of other components should be first on the list.

3. Identify the components

   Identify the names, version numbers, and, as appropriate, languages of the individual components that you need to migrate.

4. Migrate the components using the batch and online compilers and utilities

These steps are discussed on the following pages.

Before You Begin

Before you begin a migration, you may want to back up all involved files. These files can include:

- Source and target DDLDML, DDLDCLOD, DDLCAT, DDLCATX, and DDLCATLOD areas
- Source and target source libraries
- Source and target load libraries
- Source and target JCL procedure libraries

These backups provide coverage during the migration as well as after the migration is complete. If problems arise at any time, you can restore individual components or entire files from the backups.

Step 1: Determine the Types of Components to Migrate

The components to be migrated should include not only what needs to be migrated but also what is affected by the migration. The descriptions that follow identify components typically involved in migration and how these affect other components.

CA ADS Application Structure

The application structure is saved as a load module in the DDLDCLOD area of the data dictionary; no source definitions for the application are stored in the DDLDML area. The application structure is relatively autonomous. If you make changes to the application structure, you do not need to recompile any other application components.

Changes to the application structure, however, can logically affect other components, specifically dialogs. For example, if you change a response name, you will want to change the response field value of any response processes you expect to execute before control is passed to the response. The application will execute without modifying the dialog, but it will not produce the expected results.

Maps

Changes to maps fall into two categories:

- Critical changes
- Noncritical changes

Critical changes update the date/time stamp. Any dialogs that use the map must be recompiled before they can be executed. Critical changes to maps include:

- Adding a data field to the map
- Deleting a data field from the map

Noncritical changes to maps do not cause the map date/time stamp to be updated and, therefore, do not affect any other application components.

Dialogs

Dialogs associate subschemas or access modules, maps, and process code. The dialog load module contains executable process source code. Recompiling a dialog containing SQL statements creates a new relational command module (RCM). Any access modules that include that RCM must then be recompiled also. Recompiling a dialog does not affect any other application component.

Process Source Code

Process source code is stored in the data dictionary. Process code is compiled by the dialog compiler and becomes executable when the dialog is compiled. To have changes to process source code reflected in the dialog load module, you must recompile the dialog.

RCMs (SQL DML Applications Only)

If a program/dialog containing SQL statements is recompiled, an RCM is automatically created for it and stored as a load module in the DDLDCLOD area. If the program/dialog load module is copied intact to the production system, the RCM load module must also be copied.

Subschemas (Navigational DML Applications Only)

If you change a subschema associated with a dialog, map, or program, you do not need to recompile the dialog, map or program. If the subschema changes cause you to change the logic of a process module, you will need to recompile the dialog(s) in which the module is used. If the subschema changes affect the lengths of data elements or records or the procedural code in a program, you will need to recompile and relink the program.

Access Modules (SQL DML Applications Only)

Access modules must be compiled from scratch using the catalog that defines the database to be accessed. They *cannot* be copied in load module form like other application components. A typical migration would copy the RCM load modules, apply any needed database definition changes and then create all access modules used by the application, using the CREATE ACCESS MODULE command.

Non-SQL Data Definitions

Non-SQL data is defined in records that consist of record elements. Records are either database records, which are included in a schema, or work records, which are defined through the DDDL compiler.

Changes to database records require that all subschemas that use those records be recompiled. All SQL access modules that reference those records must also be recompiled for SQL applications that access non-SQL defined databases. Changes to either database records or work records may require map and/or dialog recompilation.

Some changes to database records require some form of restructuring to incorporate those changes into the existing database.

Note: For more information about modifying the schema definition of a non-SQL defined database, see Chapter 33, “Modifying Schema Entities".

SQL Data Definitions

Data is defined in tables that consist of columns. Changes to these tables require that all access modules that use those tables be recreated. Depending on the definition of a particular access module, this recreation may occur automatically or may have to be initiated manually. These changes may require map and dialog recompilation.

Some changes to table definitions require some form of restructuring to incorporate those changes into the existing database.

Note: For more information about modifying the schema definition of an SQL defined database, see Chapter 30, “Modifying Schema, View, Table, and Routine Definitions".

Adaptive Query Management

Adaptive query management is a feature of the IDMS SQL option that automatically recompiles access modules in response to certain kinds of changes in a database application. For example, if a dialog/program has been recompiled, the runtime SQL engine detects whether corresponding access modules have been recompiled to include the new RCM. If not, it automatically recompiles the access module at runtime (if the AUTO RECREATE ON option was specified when the access module was created or last altered). Adaptive query management applies to SQL DML applications that access either non-SQL or SQL-defined databases.

Adaptive query management also automatically recompiles existing access modules that access SQL-defined databases when the definitions of those databases change. Note that this does *not* happen for non-SQL defined databases. It is the responsibility of the applications administrator to manually recompile any access modules affected by changes to a non-SQL defined database.

Edit and Code Tables

Changes to stand-alone edit and code tables that are associated with a map require that the map be recompiled only if the tables are *linked* to the map. Changes to unlinked tables do not affect the map load module.

Examples

Example 1––Adding a Data Item to a Screen

Suppose users of an application request an additional data item on a screen. To determine what is affected, consider the relationship between the map and the new data item:

For an application using navigation DML, you need to do the following if the data item is from a database record already being used by the map:

1. Change the map to display the data item
2. Recompile the map
3. Recompile any dialogs that use the map

   To take these actions, you need to migrate the map and the dialogs.

For an application using navigation DML, you need to do the following if the database record is part of the subschema used by the map, but the record is not already in use by the map:

1. Add the record to the map definition
2. Change the map to display the data item
3. Recompile the map
4. Recompile any dialogs that use the map

   To take these actions, you need to migrate the map and the dialogs.

For an application using navigation DML, you need to do the following if the database record is not already part of the subschema:

1. Add the record to the subschema
2. Recompile the subschema
3. Add the record to the map definition
4. Change the map to display the data item
5. Recompile the map
6. Recompile any dialogs that use the map

   To take these actions, you need to migrate the subschema, map, and dialogs.

If the data item can be derived (for example, calculated) from data already available to the application, you need to:

1. Create a work record for the map and add it to the map definition or modify the existing work record
2. Change the map to display the data item
3. Recompile the map
4. Change any processes that must derive the data item
5. Recompile any dialogs that use the map

   To take these actions, you need to migrate the record, subschema, map, affected processes, and dialogs.

If the application uses SQL DML, a work record will already have been defined to move data between the map and the SQL statements in the dialog. To add another database item to the screen, you need to:

1. Add the item to the work record already defined for the host variables referenced in the SQL DML statements.
2. Change the map to display the data item.
3. Recompile the map.
4. Make necessary changes to the SQL statements to retrieve the data item from the database.
5. Recompile any dialogs that contain altered SQL statements and any dialogs that use the map.
6. Recompile (using the ALTER ,kACCESS MODULE statement) any access modules that contain the recompiled dialogs.

Example 2—Implementing a New Application

Suppose you implement an entirely new application based on an existing database. When the new application has been adequately tested, all of the application components need to be migrated from the test system to the production system. In addition, you must also consider what database components to migrate:

- If you have not made any changes to the structure of the database, then the existing schema and physical definitions are not affected
- Depending on the volume and type of activity involved in the new application, you may need to adjust the buffers and review the adequacy of the journals in the global DMCL
- If the application uses navigational DML and you used existing subschemas, they, too, are unaffected by the migration. However, if you created new subschemas for the application, you must migrate them.
- If the application uses SQL DML, you must migrate any RCMs and access modules that were created as part of the application.

Step 2: Determine the Sequence of Migration

Can Migrate Load Module at Any Time

If you choose to migrate only load modules, the sequence in which you migrate them does not matter.

Sequence Matters for Source Code Migration

If you migrate any source code, the sequence is *very* important because there are dependencies among the components.

In some migrations, certain components will already be in place; in others, you will need to migrate all components. The list below shows the sequence required if all components were to be migrated.

Non-SQL database definitions

1. Elementary data items
2. Group level data items
3. Database records
4. Schemas
5. Subschemas

SQL database definitions

1. Schemas
2. Tables
3. CALC keys
4. Indexes
5. Constraints

Physical database definitions

1. Segments
2. Areas
3. Files
4. DMCL modules
5. Database name tables

Application components definitions

1. Edit and code tables
2. Work records for elementary data items,group level data, maps, and dialogs
3. CA ADS process modules
4. Modules called by CA ADS processes or other programs
5. Maps
6. CA ADS application structures
7. CA ADS dialogs
8. RCMs (for SQL DML applications only)
9. Access modules (for SQL DML applications only)

Components that can be migrated in any sequence

1. Load modules
2. Source code for batch and online programs
3. CA Culprit source code
4. JCL

Step 3: Identify the Individual Components

Having determined the types of components you need to migrate, you can begin to identify the individual occurrences. To identify them uniquely, you need both their names and version numbers. For modules, programs, and edit/code tables, you also need the name of the language in which they are programmed.

Step 4: Migrate the Components

Depending on the volume of information and the configuration of your dictionaries, you can use batch or online facilities to move or copy the component definitions to their target dictionary.

Using Online Compilers for Migration

If the volume of information is small and both dictionaries are under the control of the same CA IDMS/DB or DC/UCF system, you can use the online compilers for most of the migration.

Using Batch Compilers for Migration

If the volume is large or if the dictionaries are under the control of separate CA IDMS/DB or DC/UCF systems, you need to use the batch compilers and utilities.

Migrating Only Load Modules

If you only want to create an executable application in the production environment, you migrate just the essential load modules. Note that for SQL DML applications, the access modules must still be compiled from scratch on the production system.

Migrating the Complete Application

If you want a complete, fully documented application in the production environment, you need to:

- Migrate the source for all components
- Recompile the components
- Recompile the corresponding load modules

Identification Aids

The descriptions below identify facilities or techniques you can use to identify the individual application components you need to migrate. To extract information on components stored in libraries or other data sets, use an appropriate operating system utility.

IDD DISPLAY Statement

Using either the online or batch dictionary compiler, you can list the names and version numbers of entity occurrences with a simple form of the DISPLAY ALL statement. Any of the IDD entity types can be displayed.

Using an optional WHERE clause on the DISPLAY ALL statement, you can more closely select the occurrences you want displayed. With any entity types, you can qualify the occurrence name. For some entity types, there are additional selection criteria that you can specify, such as the user ID of the person who created the entity.

Note: For more information on the DISPLAY ALL statement and its WHERE clause, see the discussion on entity-occurrence display in *CA IDMS IDD DDDL Reference Guide.*

Command Facility

With either the online or batch command facility, you can:

- Display physical database definitions
- Use a SELECT statement to list, but not display the syntax of, SQL entity definitions

IDMSRPTS

IDMSRPTS is a utility that produces reports on information stored in the dictionary. One of its options, the Program Cross-Reference Listing, is particularly useful for migration operations if you are using program registration. The report lists all subschemas for a specified schema and all of the programs registered against those subschemas.

Note: For a sample of this report and instructions on how to run the IDMSRPTS utility, see the *CA IDMS Utilities Guide*.

DREPORTs

DREPORTs also report on information stored in the dictionary. There are some DREPORTs that summarize information for dictionary entities and some that present detailed information on these entities.

From the summary reports, you can obtain the names and version numbers of the components that need to be migrated. If you need to know whether other related components will be affected, you can run one or more of the reports that present detailed information.

Note: For more information about DREPORTs, see the *CA IDMS Reports Guide*.

AREPORTs

AREPORTs report on CA ADS dialogs, application structures, and their associated components (such as subschemas, RCMs, maps, and processes) from the information stored in the DDLDML area of the dictionary.

The complete detail report is most useful when you are planning the migration of an entire application. When planning the migration of more than one dialog, run the report that keys in on only the dialogs you need.

Note: For more information about AREPORTs, see the *CA IDMS Reports Guide*.

SQL Catalog

The SQL catalog contains the definitions of all SQL-defined database entities. It also contains information on all access modules and the tables that they reference (or records, for SQL DML applications that access non-SQL defined databases). Since the catalog is itself an SQL-defined database, SQL SELECT statements may be used to query its contents.

Dictionary Classes and Attributes

Classes and their attributes are primarily a means of extending the documentation capabilities in the dictionary. When migrating, documentation by class and attribute provides a powerful mechanism to analyze and identify the components involved. Using classes and attributes provides you with the capability to display a simple list of names or to report on the details of all components having the same attribute. For example, using the DDDL compiler, you can display all modules associated with attribute TEST within class STATUS:

```
display attribute test within class status with modules.
```

Note: For more information about creating classes and attributes and display entities based on class and attribute, see the *CA IDMS IDD DDDL Reference Guide*. For more information about reporting by class and attribute, see the *CA IDMS Reports Guide*.

Naming Conventions

Naming conventions help in identifying and migrating components.

Although there are no hard-and-fast rules for designing naming conventions, there are a few factors that you should keep in mind:

- Collating sequence

  Many of the DREPORTs display the components sorted in ascending order by name. If the names of all components of an application begin with the same few characters, it is easy to distinguish one application from another, but more difficult to distinguish components within an application. Likewise, if the names of all elements within a record begin with the same few characters, it is easy to distinguish one record from another in a list, but more difficult to distinguish elements within a record.

- Acceptable name lengths

  The software permits names of different lengths for different components. If you want several characters of every name to identify the application, select a small number (for example, 2 or 3) of characters for this purpose, in order to leave enough characters for other purposes.

- Consistent number of characters

  Consider selecting a consistent number of characters to identify the record in which an element is placed or components within a particular application. If you choose a standard number of characters and place them in a standard position, it will be easy to sort information or to scan lists or reports for a particular item.

As an alternative to embedding an application identifier in component names, you may choose to use a class/attribute pair. This arrangement allows more characters per name for other purposes, while still providing a connection between components of the same application.

Migration Tools

Most of the compilers and utilities you use to create database and application components also have options that support migration. The table below summarizes these tools:

Component	Tool	Task Code	Batch Program
Non-SQL defined schema source	Schema compiler	SCHEMA	IDMSCHEM
SQL-defined schema source	Command facility	OCF	IDMSBCF
Physical database definitions ■ Segments, areas, files ■ Database name tables ■ DMCL source and load modules	Command facility	OCF	IDMSBCF
Subschema source	Subschema compiler	SSC	IDMSUBSC
Subschema load module	DDDL compiler 1; subschema compiler	IDD; SSC	IDMSDDDL; IDMSSUBC
Definitions of: ■ Elements ■ Messages ■ Modules ■ Programs ■ Records	DDDL compiler 1	IDD	IDMSDDDL
Edit/code table source	DDDL compiler 1	IDD	IDMSDDDL
Map source	Mapping utility Mapping compiler		RHDCMPUT RHDCMAP1
Module source ■ Copybook-style modules ■ CA ADS process code	DDDL compiler 1	IDD	IDMSDDDL

Component	Tool	Task Code	Batch Program
Load modules for: ■ Applications ■ Dialogs ■ Maps ■ Edit/code tables ■ RCMs	DDDL compiler 1	IDD	IDMSDDDL
Access modules	Command facility	OCF	IDMSBCF

1. All definitions that can be migrated using the DDDL compiler can also be migrated from the command facility.

General Methods

Migration Tasks

Migration generally consists of two or three tasks:

- Punching or decompiling components from a dictionary to a temporary work file or external file
- Compiling the components from the temporary work file or external file into the target dictionary
- Recompiling load modules, as necessary, in the target dictionary

The options of the schema, subschema, DDDL compilers, and command facility that you use for these tasks function identically. Different options exist in the mapping compilers and the mapping utility, and CA ADS compilers.

The following discussions explore the methods of migration using the DISPLAY and PUNCH statement options of the schema, subschema, and DDDL compilers and the command facility, and the various parameters of the mapping compilers and the mapping utility.

Using the DISPLAY statement

Use for Small Volumes of Data

The DISPLAY statement of the schema, subschema, and DDDL compilers, and command facility is useful for moving small volumes of information between dictionaries under the control of the same DC/UCF system. Because this technique occurs online, system resources, such as response time and storage pool space, will limit the volume you are able to migrate.

Steps

There are four steps in the technique:

1. Sign on to the dictionary containing the components (the source dictionary)
2. Display the individual components using the AS SYNTAX clause.

   This step accomplishes the task of decompiling the components to a temporary work file. If you need to modify existing components in the target dictionary, use the VERB MODIFY option of the DISPLAY statement (DISPLAY ADD is the default action):

   ```
   display subschema empss01 as syntax verb mod.
   ```
3. While the components are in the compiler's work file, insert a SIGNON statement for the target dictionary into the work file as the first statement.

   This step prepares for the task of compiling the components from the temporary work file into the target dictionary. At the conclusion of this step, the work file contains a SIGNON statement for the target dictionary, followed by ADD or MODIFY statements for those components you want to migrate.

   Note: Typically the output of the previous step includes an echo of the input, so the first statement in the output is the DISPLAY statement. The DISPLAY statement is not necessary, so you can replace it with the SIGNON statement.
4. Invoke the compiler

   The compiler signs you off the source dictionary, signs you on to the target dictionary, and adds or modifies the components in the work file.

Final Tasks for Schemas and Load Modules

This technique will copy the source to the target dictionary, but it does not automatically validate schemas or recompile load modules for subschemas and edit and code tables. You can perform these additional functions in one of two ways:

- After you compile the source into the target dictionary, establish currency on the appropriate component and issue the VALIDATE or GENERATE statement. To establish currency, issue a simple MODIFY statement for the component. For example:

  ```
  modify subschema empss01.
  generate.
  ```

- Before you compile the source into the target dictionary, edit the work file by inserting the VALIDATE or GENERATE statement after the source for the component.

Using the PUNCH Statement

Used for Batch Migration

The PUNCH statement of the schema, subschema, and DDDL compilers and command facility is useful for batch migrations. If you perform the migration in batch mode, the PUNCH statement allows you to migrate larger volumes of information. It also allows you to migrate between dictionaries under the control of different DC/UCF systems.

Writes Information to File or Module

The PUNCH statement has the same options as the DISPLAY statement. However, it writes the requested information to one of two destinations: an external file or an IDD module.

Use Files or Modules to Accumulate Large Numbers of Components

The file or module provides an intermediate place to store the information you want to migrate. As a result, you can:

- Accumulate components in one or more modules over the course of several terminal sessions
- Accumulate several files of components over the course of separate executions of the batch compiler
- Edit the content of the modules or files; For example, to change the STATUS of components from TEST to PRODUCTION

Technique 1

This technique is very similar to the technique for the DISPLAY statement described above. Because it occurs in batch, however, you can migrate larger volumes of information.

Steps

The steps in this technique follow:

1. In the first execution of the compiler, sign on to the source dictionary in batch mode and punch the individual components to an external file.

   In the PUNCH statement, use the AS SYNTAX clause. In addition, specify VERB MOD if you are migrating existing components. Define the file as SYSPCH in the JCL.

   To avoid having to specify these clauses in every PUNCH statement, you can issue a SET OPTIONS statement before the PUNCH statements:

   ```
   set options display as syntax verb mod.
   ```

2. When the job ends, edit the external file as follows:

   - Insert a SIGNON statement for the target dictionary as the first statement.
   - Insert the following statement after the SIGNON statement:

     ```
     set options input 1 thru 80.
     ```

     This step prepares for the task of compiling the components from the temporary file into the target dictionary. Be sure the SIGNON and SET OPTIONS statements start between columns 1 and 72.

   - Execute the compiler a second time, using the edited file as input.

     The compiler signs on the target dictionary and adds or modifies the components in the file.

Technique 2

With this technique, you create a dictionary module in the source dictionary to hold the components you want to migrate. You migrate the module to the target dictionary, extract the ADD or MODIFY statements for the individual components, and store or modify each of the components in the target dictionary.

Steps

The steps in this technique follow:

1. In batch or online mode, sign on to the source dictionary and create a module occurrence to hold the components to be moved. For example:

   ```
   add module holdit.
   ```

2. While signed on to the source dictionary, punch the components to be moved into the module using the TO MODULE and AS SYNTAX clauses:

   ```
   punch element emp-last-name
     to module holdit
     as syntax.
   ```

The module source for HOLDIT now consists of the ADD ELEMENT EMP-LAST_NAME statement.

You can perform this step in batch or online mode, and you can punch more than one component to the module. If you use the SET OPTIONS statement following signon, your input appears as follows:

```
set options input 1 thru 80
            default is on
            punch to module holdit
            as syntax.
punch element emp-last-name.
  .
  .
  .
```

This statement automatically changes an ADD to MODIFY if the entity already exists in the dictionary and punches the entity as syntax.

3. In batch mode, sign on to the source dictionary and punch the module to an external file.

   The input to the compiler consists of only two statements: a SIGNON statement for the source dictionary and a PUNCH statement for the module. In the PUNCH statement, use the AS SYNTAX and TO SYSPCH clauses. Also, be sure to define the file as SYSPCH in the JCL.

   At the end of this step, the external file contains only one statement: an ADD/MODIFY MODULE statement. Within the MODULE statement, however, the module source consists of the ADD or MODIFY statement for the individual components that you want to migrate.

4. Edit the external file as follows:

   - Insert a SIGNON statement for the target dictionary as the first statement
   - Insert the following statement after the SIGNON statement:

     ```
     set options input 1 thru 80.
     ```

   - Insert an INCLUDE MODULE statement as the last statement.

   As a result of the editing, the external file contains four statements:

   - A SIGNON statement
   - A SET OPTIONS statement
   - An ADD MODULE or MODIFY MODULE statement
   - An INCLUDE MODULE statement.

For example:

```
  signon user dba password pass dictname target.
  set option input 1 thru 80.
add module holdit
 .
 .
 .
module source follows
add element emp-last-name
  version is 1
  pic is x(20)
 .
 .
 .
msend.
include module holdit.
```

5. Execute the compiler in batch mode, using the edited file as input.

   The compiler signs on to the target dictionary and adds or modifies the module. The INCLUDE statement brings the module source into the compiler's work file. The content adds or modifies the individual components to the target dictionary.

Final Tasks for Schemas and Load Modules

As with the DISPLAY statement, the PUNCH statement does not automatically validate the schemas or generate the load modules for subschemas and edit/code tables. To perform these function, use one of the methods described earlier in 26.6.1, "Using the DISPLAY statement".

Technique 3

This technique combines parts of the Technique 2 presented above and parts of the online DISPLAY technique described earlier in 26.6.1, "Using the DISPLAY statement". Because this technique entails an online migration, you need to moderate the volume of information you punch.

Steps

The steps in this technique follow:

1. In online mode, sign on to the source dictionary and create a module occurrence to hold the components to be moved.
2. While signed on to the source dictionary, punch the components to be moved to the module.

   As above, direct the output of the punch to the module by including the TO MODULE clause in each PUNCH statement or in a SET OPTIONS statement. Also, specify the AS SYNTAX clause and the VERB ADD or VERB MODIFY clause, as appropriate.
3. Clear the compiler's work file.
4. Display the module.

   This step brings the module (with all of its source) into the compiler's work file. In the DISPLAY statement, use the AS SYNTAX clause.
5. Edit the work file as follows:
   - Insert a SIGNON statement for the target dictionary as the first statement.
   - Insert an INCLUDE MODULE statement as the last statement.

   This step prepares the work file for the task of compiling the module and then the components into the target dictionary. As a result of the editing, the work file contains three statements:
   - A SIGNON statement
   - An ADD MODULE or MODIFY MODULE statement
   - An INCLUDE MODULE statement
6. Invoke the compiler

   The compiler signs on to the target dictionary and adds or modifies the module. The INCLUDE statement brings the module source into the compiler's work file and executes the content of the work file. The content adds or modifies the individual components to the target dictionary.

Final Steps for Schemas and Load Modules

As with the other techniques, this technique does not automatically validate schemas or generate load modules for subschemas and edit/code tables. To perform these functions, use one of the methods described earlier in 26.6.1, "Using the DISPLAY statement".

Using the Mapping Compiler and Mapping Utility

Steps

There are three steps to migrate maps between dictionaries (whether under the same CA IDMS/DB or DC/UCF system or not):

1. Decompile the maps from the source dictionary.

   For this step, use the decompile function of the mapping utility (RHDCMPUT). You can decompile one or several maps in a single execution:

   ```
   PROCESS=DECOMPILE
   MAP=map1-name
   MAP=map2-name
     .
     .
     .
   ```

   The output of the decompilation consists of the source form of the maps, typically stored in a temporary file.

2. Compile the maps into the target dictionary.

   For this step, use the file of decompiled maps from the previous step as input to the mapping compiler (RHDCMAP1). The mapping compiler places a source description of the map in the DDLDML area of the target dictionary.

3. Generate the load modules for the maps in the target dictionary.

   For this step, use either the online mapping facility or the load function of the mapping utility (RHDCMPUT). If you use the load function of the mapping utility, you can generate multiple load modules in a single execution:

   ```
   PROCESS=LOAD
   MAP=map1-name
   MAP=map2-name
     .
     .
     .
   ```

Specify Source and Target Dictionary

The source and target dictionaries are typically part of multiple dictionary environments. Consequently, you must indicate which of the dictionaries the mapping compiler and mapping utility should run against. There are several techniques for specifying a particular dictionary in a multiple dictionary environment.

Note: For more information, see Chapter 25, “Dictionaries and Runtime Environments”.

For SQL-Defined Entities

Using DISPLAY and PUNCH

SQL-defined entities (schemas, tables, and so on) can be migrated using the DISPLAY and PUNCH techniques described earlier in this chapter. This approach is useful for creating a second entity that has the same definition as one in test.

Including All Related Entities

To generate syntax for all entities related to an entity whose definition is being displayed or punched, specify the FULL option.

For example, the following statement will generate syntax for all entities in the EMP schema.

```
display schema emp full as syntax
```

The next statement will generate syntax for the EMP.DEPT table and its associated indexes, constraints and calc keys.

```
display table emp.dept full as syntax
```

Replicating Physical Attributes

By default, when a DISPLAY or PUNCH statement is used to recreate the DDL for an SQL-defined entity such as a schema, the definition will be logically identical to the original. However, certain physical attributes that normally are assigned internally by the DBMS when an entity is created will not be the same if the generated DDL is used to create a new entity. These attributes include such things as a table's numeric identifier (its table ID) and the timestamps that are used to track when an entity's definition is changed.

Ensuring that physical (as well as logical) attributes are identical for all entities associated with an area or segment, facilitates copying of data from one area or segment to another. For example, you could use operating system facilities to copy production files to a quality assurance environment for testing purposes if the physical attributes of both are identical.

Note: To copy files this way, the definitions of the segments in which the files are defined must also be identical except for segment name and page group.

It is possible to define new entities with the same physical attributes as those of another entity by specifying the FULL PHYSICAL clause on the DISPLAY or PUNCH statement. This will cause additional syntax to be generated that will explicitly establish values for the physical attributes of the new entity.

The following entities have physical attributes that need to be considered:

- Schema
- Table
- View
- Table Procedure
- Procedure
- Function
- Index
- Area

Keeping Physical Attributes Identical

Each time an SQL-defined entity definition is directly or indirectly changed, its definition timestamp is changed automatically. For example, if a new index is added to a table, the table's timestamp is updated automatically. To keep two or more entity definitions identical, and after making the same change to the cloned definition, you must use an ALTER statement to set the cloned entity's timestamp to be the same as that of the original entity.

For example, the following procedure ensures that two tables have the same timestamp after adding an index to each of them.

1. Add the index to the first table:

   ```
   create index emp.depts on emp.dept (name)
   ```

2. Determine its new timestamp value:

   ```
   display table emp.dept with timestamp
   ```

3. Add the index to the second table:

   ```
   create index emp2.depts on emp2.dept (name)
   ```

4. Force the second table's timestamp to be the same as the first:

   ```
   alter table emp2.dept timestamp 'yyyy-mm-dd-hh.mm.ss.tttttt'
   ```

Note: For more information about SQL DDL, DISPLAY and PUNCH statements, see the *CA IDMS SQL Programming Guide*.

Area-Level Timestamps

If an area was defined with the STAMP BY AREA clause, then each time a change is made to the definition of any table stored in the area, the area's timestamp is updated. This is the timestamp that must be manipulated to maintain identical definitions. An area's timestamp is also updated whenever an associated table is created or dropped.

Additional Considerations

When to Migrate

You can perform most migration activities during regular working hours. Obviously, identifying, punching or decompiling components, and adding or modifying source definitions of components will not disturb programs or systems that are currently executing.

Perform Some Tasks After System Shutdown

Depending on the specifics of the migration, you may not have to do any of it after regular working hours. To be on the safe side, however, you should plan to migrate or recompile load modules during off-peak hours. You must also perform any restructuring operations on the production database while no application access is taking place. Note that if it is an SQL-defined database, the restructuring occurs immediately as part of the execution of the DDL statement that define the change. Therefore, you may want to delay execution of the DDL statements until a time when the database is not being accessed by application programs.

Making Load Modules Available

If you migrate or recompile new copies of existing load modules while the system is down, they automatically come into use when you bring the system back up. If you migrate or recompile existing load modules while the system is up, you can control the time at which the new load modules take effect through the NEW COPY option of the SYSTEM system generation statement or DCMT VARY PROGRAM command.

NEW COPY Options

Using the NEW COPY option of the SYSTEM system generation statement, you can designate whether new load modules should be loaded automatically by the system or manually through explicit commands. If you choose to control loading manually, issue the DCMT VARY PROGRAM command with the NEW COPY option.

If you are migrating a new system whose tasks and programs are not enabled in the system generation, then you can migrate or recompile all of its load modules at any time. Access to the load modules will not be possible until the tasks and programs are enabled.

Check Your Work

When you have completed the mechanical migration of components, run a series of reports or issue a series of DISPLAY statements to check your work. However, to verify that the migration is complete and successful, you must test the new components in their new environment.

Additional Tasks

Updating System Generation

A new application may have an impact on system generation. Minimally, it may require a new task definition. Other system resources, such as program pool and storage pool space, may also need to be adjusted.

Updating Users

New user IDs may have to be defined and existing user definitions reviewed.

Updating the Task Application Table

If you choose to recreate and recompile an application structure in a target dictionary, the recompilation automatically updates the task application table (TAT) for that dictionary. If you choose simply to migrate the load module of an application structure, you must manually update the TAT for the target dictionary.

There are two utilities for updating the TAT:

- ADSOTATU works in online mode
- ADSOBTAT works in batch mode

Note: For information about how to execute these utilities, see the *CA ADS Reference Guide*.

Backup the New Files

After you have migrated and tested the components, back up the files in the new environments.

Cleanup

The migration methods described throughout this chapter create copies of components. They do not physically move the components or automatically delete them from the source dictionary after the migration is complete.

If you decide to maintain a single copy of all components, you need to delete the unwanted copies. Be sure to delete all versions of both source definitions and load modules. Also be sure to delete copies of load modules from both the dictionary load areas and load libraries.

Chapter 27: Modifying Physical Database Definitions

This section contains the following topics:

Modifications You Can Make

Changes You Can Make and What To Do

The following tables summarize the changes you can make to physical database definitions and how to make the change. In most cases, all you need to do is:

- Alter the entity's definition
- Generate, punch, and link all DMCLs associated with the entity definition

However, if the entity is defined to the runtime DMCL, some changes affect how CA IDMS/DB processes a request to make the modified DMCL available dynamically. The following tables identify those changes.

When the tables indicate that you must unload and reload a segment or an area, you can use any of the following to accomplish this:

- The REORG utility statement
- The UNLOAD and RELOAD utility statements
- CA IDMS/DB Reorg

Note: For more information about REORG and UNLOAD/RELOAD, see the *CA IDMS Utilities Guide*. For more information about CA IDMS/DB Reorg, see the *CA IDMS/DB Reorg User Guide*.

Segment Definition

Change you can make	How to make it
■ The schema reserved for defining tables and indexes within areas associated with the segment ■ The segment's page group	Alter the segment's definition and generate, punch, and link all DMCLs to which the segment is defined
The maximum number of records or rows per page	■ Alter the segment definition and generate, punch, and link a DMCLin which the segment is included. ■ If the segment is empty, use this DMCL to execute a FORMAT SEGMENT utility statement. ■ If the segment is not empty, use this and a DMCL containing the original segment definition to execute the CONVERT PAGE utility statement. ■ Generate, punch, and link all remaining DMCLs in which the segment is included.

File Definition

Change you can make	How to make it
■ The external file name ■ The file's allocation information, such as the data set name and disposition	Alter the file's definition and generate, punch, and link all DMCLs in which the segment that contains the file is defined
The file's access method (VSAM or non-VSAM)	See 27.4, "Changing the Access Method of a File"

Area Definition

Change you can make	How to make it
Change the page range (but not the number of pages) assigned to an area	■ Alter the area's definition and generate, punch, and link a DMCLin which the area's segment is included. ■ If the area is empty, use this DMCL to format all files in the area ■ If the area is not empty, use this DMCL and a DMCL containing the original segment definition to execute the CONVERT PAGE utility statement. ■ Generate, punch, and link all remaining DMCLs in which the segment is included.
Increase the area's page size	See 27.5.1, "Increasing the Page Size of an Area"
Extend the area's page range	See 27.5.2, "Extending the Page Range of an Area"
Change the primary number of pages assigned to the area'spage range	If the area is not empty, unload and reload the area
Decrease the size of the area's pages	If the area is not empty, unload and reload the area
Increase or decrease the page reserve	■ Use an area override in the DMCL definitionfor special operations, such as loading a database; then remove the area override ■ To make a permanent change, alter the area definition; if the area is not empty, changing the page reserve affects only subsequent store and insert operations ■ Generate, punch and link the DMCL(s) that contain the area override or the segment that contains the defined area
Add, modify, or drop a symbolic definition 1	Alter the area's definition and generate, punch, and link all DMCLs in which the segment that contains the area is defined
Re-assign the area to new or different files	See 27.6, "Adding or Dropping Files Associated With an Area"

Change you can make	How to make it
Update an area's timestamp	Alter the area's definition and specify the new timestamp value. This applies only to areas defined for SQL use.

1. If changing the page range of a subarea associated with a record in a non-empty area, unload and reload the area. If changing the page range of a subarea associated with an index in a non-empty area, use the MAINTAIN INDEX utility statement to rebuild the index in the new page range as described in the *CA IDMS Utilities Guide*.

DMCL Definition

Change you can make	How to make it
■ Reassign the buffer associated with a file ■ Associate or disassociate a database name table ■ Add or remove a segment ■ Change an area's startup or warmstart status ■ Change an area's page reserve ■ Change the external file name for a file ■ Change the disposition for a file ■ Change the use of memory caching for a file ■ Change the shared cache assigned to a file	Alter the DMCL definition and generate, punch, and link the DMCL

Database Buffer Definitions

Change you can make	How to make it
■ Change the buffer page size ■ Change the buffer page count ■ Change how the CA IDMS/DB acquires storage for the buffer ■ Add or remove buffers	Alter the buffer definition and generate, punch, and link the DMCL with which the buffer is associated

Journal Buffer Definition

Change you can make	How to make it
Change the size of the journal buffer pages	See 27.7, "Changing the Page Size of a Disk Journal".
Change the number of journal buffer pages	Alter the definition of the journal buffer and generate, punch, and link the DMCL with which the journal buffer is associated.

Disk Journal Definition

Change you can make	**How to make it**
■ Change the external file name ■ Add or remove a disk journal file ■ Change the file's allocation information, such as its data setname ■ Change the number of pages in the disk journal file	■ If altering the characteristics of or removing an existingjournal file, offload its contents ■ Alter the journal file's definition and generate, punch, and link the DMCL ■ Format the journal file
■ Change the file's access method	See 27.8, "Changing the Access Method of a Disk Journal".

Change you can make	**How to make it**
■ Change the external file name ■ Add or remove a disk journal file ■ Change the file's allocation information, such as its data setname ■ Change the number of pages in the disk journal file	■ If altering the characteristics of or removing an existing journal file, offload its contents ■ Alter the journal file's definition and generate, punch, and link the DMCL ■ Format the journal file
■ Change the file's access method	See 27.8, "Changing the Access Method of a Disk Journal".

Archive Journal Definition

Change you can make	**How to make it**
■ Change the file's block size ■ Change the file's external file name ■ Add or remove archive journal files	Alter the archive file's definition and generate, punch, and link the DMCL with which the archive file is associated

Tape Journal Definition

Change you can make	**How to make it**
Change the file's external file name	Alter the tape file's definition and generate, punch, and link the DMCL with which the tape journal file is associated

Changes You Cannot Make

- Segment's type (that is, SQL or NONSQL)
- Name of a segment containing SQL tables

Making the Changes Available Under the Central Version

Most Changes Can Be Implemented Dynamically

Most physical database changes can be made effective to a CV without recycling by issuing a DCMT VARY DMCL command.

Some changes require files to be deallocated and reallocated. The ability to deallocate and reallocate files dynamically depends on the operating system and the information provided in the file definition.

Note: For more information, see 7.14, "FILE Statements".

Some changes require that the use of areas or journal files be quiesced. CA IDMS does this automatically as part of the vary operation. However, this may take a significant amount of time if long-running transactions are in progress.

Note: For more information about the impact of each type of change, see 27.3, “Dynamic DMCL Management".

JCL Considerations

If CV's execution JCL contains DD statements for files whose data set name is changed by a DCMT VARY DMCL command, remove those DD statements before cycling the system.

Restart Considerations

If using VARY DMCL to implement your changes, you must consider how to handle an unanticipated failure during and after the operation.

The recommended approach is to use change tracking so that no manual intervention is needed to restart the CV after an abnormal termination. CA IDMS automatically restarts the system correctly using the information stored in the SYSTRK files.

If change tracking is not in effect, you are responsible for maintaining a copy of the old DMCL load module. You may need to restart the CV using the old DMCL if an abnormal termination occurs while the VARY DMCL command is in progress. Additionally, you may need to update the execution JCL before restarting the system to ensure that the correct data set names are being referenced.

Note: For more information on change tracking, see "Change Tracking" in the *CA IDMS System Operations Guide*.

Making Disk Journal File Changes

If making changes to disk journal files, do not change or replace all files at the same time. For example, you cannot change the dataset name or number of pages of all journal files with a single DCMT VARY DMCL command. To accomplish this, implement the changes in two separate operations, changing only some of the journal files each time.

Because of this restriction, you cannot change the page size of the disk journal files dynamically since all journal files must have the same page size. To change the journal page size, you must shutdown and restart the CV using a DMCL with the new journal buffer page size.

Data Sharing Considerations

In a data sharing environment, most changes to an area or its associated files will not take effect until the area is varied offline in all group members in which it is shared since most area (and associated file) characteristics must be identical across all sharing members. For a list of these characteristics, see Sharing Update Access to Data.

The following procedure is recommended for making shared area or file changes:

- Modify and generate a new DMCL for all affected members
- Vary the area offline in all sharing members
- Vary a new copy of the altered DMCL in all affected members
- Vary the area online in all affected members

Note: For more information about data sharing, see the *CA IDMS System Operations Guide*.

Dynamic DMCL Management

Impact of Changes

When a DMCL is being varied, certain changes cause:

- Areas to be quiesced
- Files to be deallocated and reallocated

Change	Quiesce area?	Reallocate file?
Segment changes		
Dropping and recreating the segment	Yes	Yes
Page group	Yes	Yes
Maximum number of records per page	Yes	Yes
Segment's schema	No	No
Area changes		
Adding an area		Allocate
Dropping an area	Yes	Deallocate
Primary page range	Yes	Yes
Extending page range	Yes	Yes
Page size	Yes	Yes
Original page size	Yes	Yes
Symbolic parameters	Yes	Yes
Area-to-file mapping	Yes	Yes
Page reserve	No	No
Maximum space	No	No
Area's timestamp	No	No
File changes		
Dataset name	Yes	Yes
z/VM user id/virtual address	Yes	Yes
Access method	Yes	Yes
Disposition	No	Yes
External name (DDNAME)	No	Yes
DMCL changes		

Change	Quiesce area?	Reallocate file?
Adding a segment		Allocate
Dropping a segment	Yes	Deallocate
Buffer associated with a file	No 1	No
Shared cache usage for a file	No	No
File's external name (DDNAME)	No	Yes
File's disposition	No	Yes
Area status	No	Yes
Shared cache for a file	No	Yes
Disk Journal changes		
Adding a disk journal		Allocate
Dropping a disk journal	Yes (2)	Deallocate
Number of pages in file	Yes (2)	No
Dataset name	Yes (2)	Yes
File's external name (DDNAME)	No	Yes

1 If a file is associated with a new buffer, the area's pages are first purged from the buffer pool.

2 Use of the disk journal file is quiesced.

Considerations

- Changing the page size of a buffer causes the buffer to be closed and re-opened with the new size. All other buffer changes (such as the number of pages) are ignored. To change these parameters while the system is active, issue a DCMT VARY BUFFER command.
- Changes to a journal buffer, tape, or archive journal file have no impact on the runtime system.
- In a data sharing environment, if an area is shared, most changes to the area and its associated files will not take effect until the area is varied offline in all group members in which it is shared.

Changing the Access Method of a File

Procedure

You can change the format of database files from non-VSAM to VSAM and vice versa. To complete this process, you need to:

1. Expand the page size of the file's area, if necessary
2. Alter the file definition to change its access method (and optionally to specify a new database name or other location information) and generate, punch, and link all DMCLs in which the file's segment is included.
3. Allocate a new VSAM or non-VSAM data set, as described in Allocating and Formatting Files.
4. Make the area to be processed unavailable for update under the central version.
5. Copy the old VSAM or non-VSAM file to the new data set.
6. Make the new DMCLs and file available to the runtime environment.

Steps 1 and 5 are discussed next.

Step 1: Expand the Page Size

Converting from Non-VSAM to VSAM

When you convert a non-VSAM file to VSAM, expand the area's page size first if the page size of the area is significantly smaller than the size of the VSAM control interval. The optimal page size is 8 bytes less than the VSAM control interval.

Converting from VSAM to non-VSAM

When you convert a VSAM file to non-VSAM, consider expanding the area's page size either before or after the conversion if the page size of the area is inefficient for the device type.

Note:

- For optimal page sizes based on device type, see the *CA IDMS Database Design Guide*.
- For the steps involved in expanding the page size of an area, see 27.5.1, "Increasing the Page Size of an Area".

Step 5: Copy the Data to the New File

Options

To copy the data, use one of the following options:

1. Use the BACKUP and RESTORE utility statements
2. Use the IDCAMS utility

Option 1: Backup and Restore

To use BACKUP and RESTORE to copy the database files, take the following steps:

1. Offload the data in the old file(s) using the BACKUP utility statement and the old DMCL. If all files within a multi-file area are being converted, use the AREA option on the BACKUP statement; otherwise, use the FILE option.
2. If the backup was performed with the AREA option, format the new files using the new DMCL before executing Step 3.
3. Reload the data into the new file(s) using the RESTORE utility statement and the new DMCL. If the data was offloaded with the AREA option, restore with the AREA option; otherwise, restore with the FILE option.

Option 2: Using IDCAMS

The REPRO command of the IDCAMS utility can be used to copy the data between a VSAM and non-VSAM file and vice versa. If you use this approach, be sure to copy all pages (blocks) in the file in their entirety without reblocking.

Note: For more information about IDCAMS, see the appropriate IBM documentation.

Increasing the Size of an Area

Available Options

To increase the size of an area, you can:

1. Increase the page size of the area by using the EXPAND PAGE utility statement
2. Extend the number of pages in the area by using the EXTEND SPACE clause of the AREA statement
3. Increase the current number of pages assigned to the area by unloading and reloading the area

Which Option to Use

Both options 1 and 3 distribute free space throughout an area. While option 1 is faster (and therefore less disruptive) than option 3, it does not reorganize indexes or improve the placement of existing data which may have overflowed due to lack of space on a page. Option 1 is most effective if used before the area approaches a full condition.

Option 2 adds free space only at the end of an area. This can be useful where records or tables have a location mode of direct or are clustered around a dbkey index or an OOAK record. It can also be used as a temporary means of increasing space in an area whose page size cannot be increased (due to device or VSAM restrictions).

If the area to be extended contains CALC records, these records will continue to only target pages in the original page range. If no space is available to hold the new occurrences, they will overflow into the extended page range. The area must continue to be defined as being extended until the records are unloaded and reloaded into a new database in which the entire extended page range is defined as the original page range. Failure to do this results in 0326 errors when CALC retrieval is attempted.

In order to extend an area, there must be unassigned page numbers following the current page range. If these page numbers are already assigned to another area, the current page range cannot be extended. Either Option 1 or 3 must be used; or the current page range must be converted to a new range and that range extended.

Procedures

Procedures for the first two options follow.

Note: For information about unloading and reloading an area, see the *CA IDMS Utilities Guide*.

Increasing the Page Size of an Area

Steps

To increase the page size for an area, follow these steps:

1. For each file associated with the area, allocate a new file having the larger page size.
2. Copy the contents of each existing file to its new counterpart by executing the EXPAND PAGE utility statement once for each file.
3. Delete the old files and rename the new files to the old names.
4. Alter the area definition by removing the WITHIN FILE clause and replacing it with REMOVE FILE.
5. Alter the area by:
   - Changing the page size
   - Adding the ORIGINAL PAGE SIZE clause
   - Re-adding the WITHIN FILE clause
6. Generate, punch, and link all DMCLs in which the file's segment is included

Extending the Page Range of an Area

Steps

To extend the number of pages in an area, follow these steps:

1. If the additional pages being added to the area will reside in a new file, define the file.
2. Change the definition of the area specifying the number of additional pages to add to the area by using the EXTEND SPACE clause. On the EXTEND SPACE clause, specify to which file the additional pages will be mapped by using the WITHIN FILE clause.

   If the additional pages would cause the number of pages in the area to exceed the maximum space allowed, you can use the MAXIMUM SPACE clause to increase the maximum provided the page numbers are not assigned to another area that will be used in the same DMCL as the area being expanded.
3. Generate, punch and link all DMCLs that contain the segment with which the area is associated.

4. Allocate a new database file to contain the additional pages and initialize the file using the new DMCL.
5. If new pages are being added to the last file of the area:
   - Make the area to be processed unavailable for update under the central version.
   - Backup the area using an old DMCL.
   - Restore the area using an old DMCL, but referencing the new file through JCL statements.

     Note: If the area maps to its file on a one-to-one basis, it is necessary to include IDMSQSAM=ON in the RESTORE utility's SYSIDMS file.
   - Delete the old file and rename the new file.
   - Backup the expanded area.
6. Make the DMCLs and the new file available to the runtime environment.

Adding or Dropping Files Associated With an Area

Types of Changes

The pages of an area can be mapped to different files provided that all the pages are accounted for. For example, two files can be combined into one file or one file can be separated into multiple files.

Steps

To add or remove files from an area, follow these steps:

1. Define the new files.
2. Change the definition of the area by excluding all files associated with the area and re-assigning the pages of the area to file blocks.
3. Drop all unused files.
4. Generate, punch and link all DMCLs that contain the segment with which the area is associated.

5. Allocate and format new database files.
6. Make the area to be processed unavailable for update under the central version. (If re-using some of the existing files, take the area offline to the central version.)
7. Backup the area using the AREA option of the BACKUP utility statement and the old DMCL.
8. Restore the area using the AREA option of the RESTORE utility statement and the new DMCL.
9. Make the DMCLs and the new files available to the runtime environment.

Changing the Page Size of a Disk Journal

Steps

To change the page size of a disk journal, follow these steps:

1. Change the size of the journal buffer page.
2. Generate, punch, and link the DMCL.
3. Shut down the system.
4. Offload all currently used journals using the ARCHIVE JOURNAL utility statement with the ALL option and the old DMCL.
5. Allocate and format new disk journal files.
6. Restart the system with the new DMCL and the new journal files.

Changing the Access Method of a Disk Journal

Steps

You can change the access method used for a disk journal file from non-VSAM to VSAM or vice versa. To do this you must:

1. Change the definition of the disk journal file specifying the desired access method. Alter the page size of the journal buffer:
   - If changing from non-VSAM to VSAM, the page size should be 8 bytes less than the control interval size
   - If changing from VSAM to non-VSAM, choose an optimal page size for the device type
2. Generate, punch, and link the DMCL.
3. Shut down the system.
4. Offload all currently used journals using the ARCHIVE JOURNAL utility statement with the ALL option and the old DMCL.
5. Allocate and format new disk journal files.
6. Restart the system with the new DMCL and the new journal files.

More Information

- For more information about segment, area, and file definition, see Chapter 4, “Defining Segments, Files, and Areas".
- For more information about DMCL, database buffer, journal buffer, and journal file definition, see Chapter 5, “Defining, Generating, and Punching a DMCL".
- For more information about the syntax for physical database entities, see Chapter 7, “Physical Database DDL Statements".
- For more information about DCMT commands, see the *CA IDMS System Tasks and Operator Commands Guide*.
- For more information about utility statement syntax, see the *CA IDMS Utilities Guide*.
- For more information about data sharing, see the *CA IDMS System Operations Guide*.

Chapter 28: Modifying Database Name Tables

This section contains the following topics:

Changes You Can Make

You can modify the following characteristics of a database name table definition:

- What databases are associated with the database name table (through the DBNAME statement)
- What segments and/or subschema mappings are associated with a database name
- Generic subschema mappings defined to the database name table
- The MIXED PAGE GROUP BINDS option setting for a DBNAME
- What database groups are associated with the database name table (through the DBGROUP statement)
- The usage option for a DBNAME

Procedure for Modifying Database Name Tables

Steps

To modify a database name table, follow these steps:

Action	**Statement**
Modify the database name, database group, and/or database name table	■ CREATE, ALTER, or DROP DBNAME ■ CREATE, ALTER, or DROP DBGROUP ALTER DBTABLE
Regenerate the database name table	GENERATE DBTABLE
Punch and link the database name table to a load library	PUNCH DBTABLE LOAD MODULE
Make the database name table available under the central version	DCMT VARY DBTABLE NEW COPY

Example

In the following example, the DBA adds a new database name to an existing database name table. After generating and punching the database name table load module, the DBA instructs CA IDMS/DB to load the updated database name table:

```
create dbname alldbs.benefits
   add segment empseg
   add segment projseg
   add segment beneseg;

generate dbtable alldbs;

punch dbtable load module alldbs;
```

After link-editing the modified database name table to a load library, make it available under the central version:

```
dcmt vary dbtable alldbs new copy
```

More Information

- For more information about defining database name tables and database names, see Chapter 6, “Defining a Database Name Table".
- For more information, syntax, and syntax rules for the DBTABLE, DBGROUP, and DBNAME statements, see Chapter 7, “Physical Database DDL Statements”.
- For more information about DCMT commands, see the *CA IDMS System Tasks and Operator Commands Guide*.
- For more information about the PUNCH utility statement, see the *CA IDMS Utilities Guide*.
- For more information about database groups and dynamic routing, see the *CA IDMS System Operations Guide*.

Chapter 29: Modifying SQL-Defined Databases

This section contains the following topics:

What You Can Modify

You can modify an SQL-defined database by:

- Adding or dropping tables
- Modifying table components
- Adding, modifying, or dropping indexes and referential constraints
- Adding, modifying, or dropping schemas
- Adding or dropping views
- Adding, modifying or dropping SQL routines
- Adding or dropping keys associated with SQL routines

Note: For more information about maintaining physical definitions, see the chapter "Modifying Physical Database Definitions".

Maintaining Identically-Defined Entities

Why Identical Definitions are Useful

Maintaining entities with identical definitions can be useful in situations such as the following:

- Taking a snapshot copy of a production database for testing purposes
- Moving a test database into production
- Implementing database segmentation so that multiple segments can be accessed through a single referencing schema and set of access modules
- Restoring a back-version of a database and its definition

How to Maintain Identical Definitions

To maintain identically defined entities, you must explicitly specify physical attributes whose values would otherwise be automatically assigned when the entity is created or altered. The physical attributes that must be explicitly specified for this purpose are:

- Numeric table identifier assigned to a table when it is created
- Numeric index identifier assigned to an index when it is created
- Synchronization timestamps associated with an area, table, view, procedure, table procedure or function

The appropriate DDL statements (such as CREATE TABLE, ALTER AREA, and so on) provide clauses for the specification of these physical attributes.

Determining the Physical Attributes of Existing Entities

You can determine the values of the physical attributes assigned to an existing entity by specifying either FULL PHYSICAL or WITH TIMESTAMP on a DISPLAY or PUNCH statement for the entity. The FULL PHYSICAL option generates syntax for all attributes of an entity including physical attributes such as table IDs and synchronization stamps. The WITH TIMESTAMP option generates only the syntax for specifying a synchronization timestamp.

If you display a schema specifying FULL PHYSICAL, syntax is generated for all entities in the schema. The syntax includes specifications for all physical attributes of those entities. A final set of ALTER statements is generated to establish the value for the synchronization timestamp for all entities that have one.

Note: For more information about these clauses and the syntax used to specify physical attributes, see the *SQL Reference Guide*.

Specifying Synchronization Timestamps

While the ability to specify physical attributes can be useful in certain situations, it should be used with care. If you change the value of a synchronization timestamp, you can disable the ability for the database engine to detect definition-based changes. This could result in data corruption if an out-of-date access module updates the database.

At a minimum, you should ensure that every version of an entity's definition has a unique synchronization timestamp associated with it. You should also be aware that while some entities, such as indexes and constraints, do not have an associated timestamp, changing their definition is, in effect, changing the definition of their associated table(s) and must also result in a unique sychronization stamp value.

If a table resides in an area that is controlled by area-level synchronization stamps, you must update the area's synchronization timestamp. Updating the table's synchronization stamp is optional but recommended. If a table resides in an area that is controlled by table-level synchronization stamps, you must update the table's stamp and cannot update that of the area.

Specifying Table and Index IDs

It is not always possible to create a table with a specific table ID or an index with a specific index ID. You are able to do so only if the value specified is not assigned to another table or index in the same area. Consequently, manipulation of physical attributes is generally only appropriate for schemas that define the entire contents of a database area or segment.

Stamp Synchronization

The SYNCHRONIZE STAMPS utility lets you compare stamps in the data area and the catalog and to update one from the other.

This utility provides an alternative mechanism for maintaining identical synchronization timestamps and may be an aid in recovery situations in which either the catalog or a data area must be restored independently of the other.

For example, suppose that you want to take a copy of a production database for testing purposes. Assuming that the definitions of both are identical except for the synchronization timestamps, you can use operating system facilities for copying the data files and then use the SYNCHRONIZE STAMPS utility to update the copied areas with the timestamps from the test catalog.

Note: For more information about the SYNCHRONIZE STAMPS utility, see the *CA IDMS Utilities Guide*.

Methods for Modifying

You can use the following methods to change an SQL-defined database:

- Single DDL statement

  You use a single DDL statement to make the change. The change takes effect immediately. For example, you use a single DDL statement when adding a check constraint.

- Multiple DDL statements

  You use multiple DDL statements to make the change. The particular SQL DDL statements you use depend on the type of change being made. For example, to change certain index characteristics (such as the order in which index keys are stored) you can use these SQL statements:

  - DROP INDEX
  - CREATE INDEX

  The change takes effect upon completion of these statements.

- Combination of DML and DDL statements

  You use a combination of DML and DDL statements to modify a definition. This method often involves dropping, redefining, and reloading a table to make the change.

  Once the data has been reloaded, the change takes effect. For example, to move a table to a new area, you use DML or utility statements to:

  1. Create a new table in the new area (DDL CREATE)
  2. Copy the rows of data to the new table (DML INSERT)
  3. Delete the existing table (DDL DROP)

Choosing a Modification Method

In some cases, you may choose the method to use. In other cases, the method is dictated by database factors such as whether the table contains data or whether it participates in a referential constraint.

Each modification is discussed in detail in the following chapters.

Inform Your Users

Some changes you make to the database will have a direct impact on your users. For example, if you drop a table or a view, users will no longer have access to the data.

Before you make a change such as dropping a table, you can use SELECT statements to determine where the entity to be changed is used. Specifically, look for:

- Views that reference the table
- Referential constraints in which the table participates
- Access modules that access the table

This indicates the potential impact the change may have and provides information about determining the best method to use to make the change.

Chapter 30: Modifying Schema, View, Table, and Routine Definitions

This section contains the following topics:

Overview

This chapter describes methods for creating, dropping, and changing schemas, views, tables, and routines.

Note: For more information about the SQL DDL statements used in the procedures in this chapter, see the *CA IDMS SQL Reference Guide*.

Maintaining Schemas

This section describes how to:

- Drop a schema
- Change a component of a schema

Dropping an Existing Schema

DROP SCHEMA Statement

To drop a schema, use an SQL DDL DROP SCHEMA statement. This removes the named schema only if no tables or views are associated with it.

CASCADE Option

If you specify the CASCADE option, you also delete:

- The definition of each table and view associated with the named schema
- The data stored in each table associated with the schema
- The definition of each referential constraint, index, and CALC key defined on the tables associated with the named schema
- The view definition of each view derived from one or more of the tables associated with the named schema
- All privileges granted on tables dropped as a result of cascade processing

Considerations

If all tables and indexes on those tables are in a segment in which no other table or index from another schema resides, then you can use the FORMAT utility to erase rows and indexes before using DROP SCHEMA. This will enable more efficient execution.

Example

In the following example, a schema and its associated tables are dropped.

```
drop schema demoempl cascade;
```

Modifying a Schema

To modify a schema, use the SQL DDL ALTER SCHEMA statement.

Considerations

Changing the default area associated with the schema does not affect existing tables.

Example

In the following example, the schema's default area is changed.

```
alter schema demoempl
   default area demoempl.emplarea;
```

Maintaining Views

This section describes how to:

- Drop a view
- Change a view definition by dropping and recreating it

Dropping a View

DROP VIEW Statement

To drop a view, use the SQL DDL DROP VIEW statement.

CASCADE Option

Use the CASCADE option if the view being dropped participates in any other view definitions. CASCADE directs CA IDMS/DB to drop the named view and all views derived from the named view.

When you drop a view (*without* CASCADE), the following definitions are removed from the dictionary:

- The view
- All privileges granted on the view

If you specify CASCADE, these additional definitions are removed from the dictionary:

- All views in which the view is referenced and all views referencing those views
- All privileges granted on views dropped as a result of cascade processing

Considerations

You must specify CASCADE if there are views defined on the view you are dropping.

Example

In the following example, the view EMP_HOME_INFO is dropped. This also drops any views derived from this view.

```
drop view emp_home_info cascade;
```

Modifying a View

To modify a view, use the SQL DDL DROP VIEW statement to drop the view and then use the SQL DDL CREATE VIEW statement to re-add the view.

Before modifying a view, you can use the SELECT SYNTAX FROM SYSCA.SYNTAX statement to display the syntax used to create a view.

```
select syntax from sysca.syntax
  where schema=HR
  and table=EMP-SALARY;
```

Note: For more information about SYSCA.SYNTAX table, see the *CA IDMS SQL Reference Guide*.

Example

In the following example, the syntax for the view EMP_HOME_INFO is displayed using the SELECT SYNTAX statement. The view is then dropped (DROP) and re-added (CREATE) with an additional column (CITY).

This SELECT SYNTAX statement:

```
select syntax from sysca.syntax
   where schema=demoempl
   and table=emp_home_info;
```

Displays this view syntax:

```
create view emp_home_info
    as select emp_id, emp_lname, emp_fname, phone
        from employee;
```

DROP VIEW AND CREATE VIEW are used to modify the view.

```
drop view emp_home_info;
create view emp_home_info
  as select emp_id, emp_lname, emp_fname, phone, city
     from employee;
```

Maintaining Tables

This section describes how to:

- Create or drop a table
- Create or drop a column
- Change column characteristics
- Add or remove data compression
- Create, drop, or modify check constraints
- Revise the table's estimated row count
- Change the table's area
- Drop the default index associated with the table

Creating a Table

CREATE TABLE Statement

To create a table, use the SQL DDL CREATE TABLE statement.

Considerations

The area in which the table's rows are to reside must be defined in the application dictionary and be accessible to the runtime environment in which the CREATE TABLE statement is issued.

Dropping a Table

DROP TABLE Statement

To drop a table, use the SQL DDL DROP TABLE statement. Use the CASCADE option if the table participates in a referential constraint or is referenced in one or more view definitions.

No CASCADE

When you drop a table (*without* CASCADE), the following definitions are removed from the dictionary:

- The table
- Its CALC key (if any)
- All indexes defined on the table
- All privileges granted on the table

Table rows and indexes are removed from the database.

With CASCADE

If you specify CASCADE, these additional definitions are removed from the dictionary:

- All referential constraints in which the table participates
- All views in which the table is referenced and all views referencing those views
- All privileges granted on views dropped as a result of cascade processing

Considerations

Using FORMAT to Erase Table Rows

If the table you want to drop is the only table in an area, it participates in no linked constraints and its indexes (if any) also reside in areas in which no other table or index resides, you can use the FORMAT utility to drop the table more efficiently:

1. Format the area(s) containing the table and indexes
2. Drop the table

Note: For more information about FORMAT, see the *CA IDMS Utilities Guide*.

Dropping All Tables in a Schema

If you want to drop all tables in a schema, use the DROP SCHEMA statement with the CASCADE option rather than dropping each table individually.

Example

In the following example, these entities are dropped: the BENEFITS table, its CALC key, all indexes defined on it, all privileges on it, all referential constraints in which BENEFITS participates, all views in which this table is referenced and all views referencing that view, and all privileges granted on all those views. In addition, all data will be deleted.

```
drop table demoempl.benefits cascade;
```

Adding a Column to a Table

ALTER TABLE Statement

With the ALTER TABLE statement, you can make the following column changes:

- Add a column
- Change a column's data type or null attribute
- Drop or change a column's default clause
- Rename a column
- Drop a column

For instructions on using the ALTER TABLE statement, see the *SQL Reference Guide*.

Considerations

Add a Column

The definition of the table is updated to include the new column definition, and the new column becomes the last column in the table. Table rows are not updated as part of the ALTER TABLE processing; instead, the column is added to an existing row only when that row is next updated.

When adding a column, if the table is not empty, you much supply a default value for the added column. You do this in one of the following ways:

- Specify that the column is to have a default value. All existing rows are then considered to have the default value for the new column.
- Allow the column to have a null value. All existing rows are then considered to have a null value for the column.

Note: For more information about choosing a value, see the *SQL Reference Guide*.

Compressed Rows

If a new column in a compressed table will be used as an index key or as a referencing column, consider placing the column near the front of the table. Otherwise, the compression potential of the table will be greatly reduced.

To do this, the table must be dropped and re-added with a new column order. When you put the rows back into the table, make sure the data is in the new column order.

Effect on Programs and View Definitions

Adding a column to a table does not impact existing programs or view definitions except under the following circumstances:

- If your host language programs include SELECT * from the table, they will receive runtime errors because of the added column.
- If a view definition includes a SELECT * from the affected table, it becomes invalid and must be dropped and recreated.

Add a Default to a Column

Allowing a column to have a default value affects only the table's definition; existing table rows are not affected.

Remove a Column's Default

If the table is populated and the column does not allow null values, every existing row must contain a value in the changed column. To ensure this, each row is *accessed and updated* if it does not contain a value for the column.

Rename a Column

A column that is named in a check constraint or a view cannot be renamed.

The definition of all referential constraints, sort keys, CALC keys and indexes in which the column participates *are updated* to show the new column name.

Drop a Column

Every row in the table is updated to remove the column value.

If a column is named in a check constraint or is part of the CALC key of a populated table, you cannot drop the column.

If you do not specify CASCADE, the column must not be one of the following types of columns:

- A column in a CALC key
- A referenced or foreign key column in a referential constraint
- An indexed column
- A sort column of a linked constraint
- Named in a view

If you specify CASCADE, how the column is used determines what other items are dropped:

- Dropping a CALC key column also drops the CALC key
- Dropping a referenced or foreign key column in a referential constraint also drops the constraint
- Dropping an indexed column also drops the index
- Dropping a sort column of a linked constraint also drops the constraint
- Dropping a column named in a view also drops the view

Change a Column’s Null Attribute

The following situations apply when you change a column's null attribute:

- When the column is part of a CALC key of a populated table, or is a referenced column in a constraint, the ALTER statement fails.
- When you change a null attribute, every row in the table is updated to add or remove the null attribute byte for that column.
- When the changed column is a sort column, every index and linked indexed constraint is automatically rebuilt.
- When disallowing nulls and the value of the column is null for a row in the table, the ALTER statement fails.

Change a Column’s Data Type

The following situations apply when you change a column's data type:

- When the column is part of a CALC key of a populated table, or is a referenced column in a constraint, the ALTER statement fails.
- When changing a column’s data type, the new data type you enter must be compatible for assignment with the original data type.
- Every row in the table is restructured to convert the column value to the new type. This might involve increasing or decreasing the length of the row.
- The ALTER statement will fail if a loss of data (such as truncation of a non-blank character or numeric overflow) would occur as part of the conversion.
- When you change data type, every index and linked indexed constraint in which the column is a sort column is rebuilt.

Maximum Row Length

Adding a column to a table or changing a column's attributes might increase the length of the table row beyond the maximum allowed.

For compressed tables, the maximum is 32760. If the new column would cause this to be exceeded, the column cannot be added to the table; instead, consider creating a second table to hold the additional information.

For uncompressed tables, the maximum depends on the page size of the area in which the table resides. If the new column would cause the length of the row to be greater than (page size - 40), then do one of the following:

- Use the EXPAND PAGE utility statement to increase the page size of the areas.

Note: For more information about EXPAND PAGE, see the *Utilities Guide*.

- Compress the table.
- Create a second table to hold the new or expanded information.

Note: The maximum length of an uncompressed row can be as much as (page size - 40); however, it is recommended that row lengths be no more than 30% of the size of the page.

Expanding Space in an Area

If an area is becoming full, consider expanding its space before increasing the length of table rows. The chapter “Modifying Physical Database Definitions” describes methods you can use to expand an area.

Adding or Removing Data Compression

Drop/Add Table

To add or remove compression, you must drop and redefine the table, as described in 30.5, “Dropping and Recreating a Table”. When redefining the table, add or remove the COMPRESS clause as desired.

Considerations

- By removing compression, the table will occupy more space in the database and may overflow a database that is already near capacity
- By adding compression, you may incur a modest increase in CPU time during subsequent DML processing of the table

Note: For more information about data compression, see the *CA IDMS Presspack User Guide*.

Adding a New Check Constraint

ALTER TABLE statement

To add a new check constraint, use the SQL DDL ALTER TABLE statement with the ADD CHECK option.

Considerations

- Adding a check constraint will *append* the new check constraint to any check constraints currently on the table
- If current data does not conform to the new check constraint, you will receive an error when CA IDMS/DB processes the ALTER TABLE command

Example

In the following example, a new check constraint is added to the BENEFITS table.

```
alter table emp.benefits
   add check ( fiscal_year > 1920 );
```

Dropping a Check Constraint

ALTER TABLE Statement

To drop a check constraint, use the SQL DDL ALTER TABLE statement with the DROP CHECK option. DROP CHECK deletes *all* check constraints associated with the table.

Example

In the following example, all check constraints associated with the BENEFITS table are dropped.

```
alter table emp.benefits
   drop check;
```

Modifying a Check Constraint

To modify a check constraint, follow these steps:

1. Drop the existing check constraint, as described above
2. Add the new check constraint, as described above

Note: Use SELECT SYNTAX from SYSCA.SYNTAX to display the existing check constraints before dropping it:

```
select syntax from sysca.syntax
 where schema='EMP' and
 table = 'BENEFITS';
```

Example

```
alter table emp.benefits
   drop check;

alter table emp.benefits
   add check ( fiscal_year > 1930 );
```

Revising the Estimated Row Count for a Table

ALTER TABLE Statement

To change the estimated row count on the table definition, use the SQL DDL ALTER TABLE statement with the ESTIMATED NUMBER OF ROWS option.

Considerations

- Changing the estimated number of rows for a table will not affect default index sizing unless you drop and re-add the index or referential constraint. The estimated number of rows is used for index calculations only if it is greater than the NUMROWS column in SYSCA.TABLE. NUMROWS is updated whenever an UPDATE STATISTICS utility statement is issued for the table or the table's area.

  Note: For more information about index calculations, see the *CA IDMS SQL Reference Guide*.

- Changing the estimated row count may affect the access paths chosen by the access module compiler for SQL DML statements that reference the table. Unlike other table modifications, though, changing the estimated row count will not cause existing access modules that reference the table to be automatically recompiled. If recompilation of selected access modules is desired, you must use the ALTER ACCESS MODULE statement to force reoptimization.

  Note: Estimated number of rows is used for optimization purposes only if the NUMROWS column of SYSCA.TABLE is 0.

Example

In the following example, the estimated row count for the EMPLOYEE table is revised.

```
alter table emp_employee
   estimated row count 750000;
```

Changing the Area of a Table

Drop/Add Table

To change the area in which the rows of a table are stored, you must drop the table and redefine it specifying the new area.

Note: For the steps and considerations involved in this process, see 30.5, “Dropping and Recreating a Table”.

Dropping the Default Index Associated with a Table

ALTER TABLE Statement

To drop the default index associated with a table, use the SQL DDL ALTER TABLE statement with the DROP DEFAULT INDEX option.

Considerations

- Do not drop the default index on a table until the CALC key, indexes, and referential constraints in which the table participates have been defined. If no other index exists on the table, an area sweep will be initiated each time one of the above components is defined.
- Dropping the default index could change the location mode of a table.
- Default indexes can be useful whenever it is anticipated that a table will be accessed without WHERE clauses specifying index or CALC keys and without joins that might use referential relationships with other tables. In short, they are useful whenever it is anticipated that the optimizer would otherwise choose area sweeps to satisfy access requests on the table. This is particularly true when it is a sparse table, since a sweep of the default index will only access data pages that contain rows of the table; whereas, an area sweep will access every page of the area.

Note: For more information about when you would choose to drop the default index, see the *CA IDMS Database Design Guide*.

Example

In the following example, the default index for the EMPLOYEE table is dropped.

```
alter table emp.employee
   drop default index;
```

Dropping and Recreating a Table

Considerations for Dropping/Adding a Table

Many types of changes can only be implemented by dropping and redefining a table. There are two major considerations involved with this process:

- Preserving the table's data
- Re-establishing the table's relationships with other tables and views

This section outlines two approaches that can be used to drop and recreate a table:

- Method 1—Uses a combination of DDL and DML statements to perform the operation
- Method 2—Uses DDL and utility statements

Considerations

Select the approach based on the size of the table and the importance of minimizing the time during which the table cannot be accessed. Consider the following:

- Method 1 requires there be enough space in the database to hold two copies of the data simultaneously. It also builds indexes and validates relationships as the data is being inserted into a new table, potentially requiring a large number of row locks and journal images.
- Method 2 reloads the data in local mode using the LOAD utility statement. Therefore, the table and all other tables in the same area cannot be accessed while the load is taking place.

For these reasons, Method 1 is more appropriate for small tables, while Method 2 is more suited for large tables.

Method 1—Using DDL and DML Statements

Steps

To use a combination of DDL and DML statements to recreate a table, follow these steps:

1. Define a new table that has the same definition as the original table except for the desired changes.
2. Define the same indexes and CALC keys for the new table as for the old (unless changes in these are desired).
3. For each referential constraint in which the original table is the *referencing* table, define a similar constraint on the new table. The new constraint must be defined with a different name and if the referenced table is not empty, it must be defined as unlinked. (The unlinked constraint may also require that an index be defined, including the foreign key of the new table).
4. For each referential constraint in which the original table is the *referenced* table, determine if the referencing table is empty. If it is, define a similar constraint with a different name in which the new table is the referenced table. If the referencing table is not empty, determine if additional indexes are needed, including the foreign key of the referencing table, to support a similar constraint defined as unlinked. If additional indexes are required, create them now.

5. For each view in which the original table is referenced (or views of those views), display the definition syntax by selecting from SYSCA.SYNTAX. Save the resulting output so the views can be recreated later.

6. Copy the data from the original table to the new table using an INSERT statement with the SELECT option.

7. For each referential constraint in which the original table is the *referenced* table and the referencing table is not empty, define a constraint in which the new table is the referenced table. The new constraint must have a different name and be defined as unlinked.

8. Drop the original table using the CASCADE option of DROP table.

9. For each *self-referencing constraint* defined on the original table, define a similar constraint on the new table. (A self-referencing constraint is a referential constraint in which the referenced and referencing table are the same.)

10. Complete the transition to the new table as follows:

    - Define a view on the new table with the same name as the original table and including all of its columns.
    - Recreate the views whose syntax was previously saved; examine those view definitions to see if changes are required.
    - Re-specify privilege definitions on the individual table and views if access is controlled through CA IDMS internal security.

Guaranteeing Integrity of the Data

Steps 6 through 8 should be performed within a single transaction to minimize the potential of changes to the data in the original table and any of its related tables until the entire operation is completed. To *ensure* that no changes are made between the time the data is copied and the time the table is dropped, take one of the following actions just prior to issuing the SELECT statement:

- Prohibit access to the table by explicitly dropping all views that reference it. This is effective only if all update access to the table is done through a view.
- Revoke all INSERT, UPDATE, and DELETE privileges from the table (and any matching wildcarded table names) if access is controlled through CA IDMS internal security.
- Alter the original table and add a dummy column. This has the effect of prohibiting access to the table until the transaction has terminated.

Recreating Empty Tables

If the table to be recreated is empty, you need not define a new table. Instead, simply drop and redefine the table making the desired changes to its definition. However, be sure to take appropriate steps to preserve referential constraints, views derived from the table, and privilege definitions.

Method 2—Using DDL and Utility Statements

Steps

To use a combination of DDL and utility statements to drop and recreate a table, take the following steps:

1. Identify all tables related through a linked constraint to the target table (the table whose definition is to be changed). Either the related tables must be unloaded and reloaded together with the target table or the constraints will become unlinked when they are redefined.

2. For each view in which the target table is referenced (or views of those views), display the definition syntax by selecting from SYSCA.SYNTAX. Save the resulting output so the views can be recreated later.

3. For each table to be unloaded, extract the data to a sequential file using either:

   - A user-written program
   - A CA Culprit report

   Use separate extract files for each table or place an indicator in each output record to identify the table from which the data was extracted. Be sure the data was extracted successfully before proceeding to the next step.

4. Drop the target table (specifying the CASCADE option) and delete the rows from the related tables that were unloaded by using a DELETE statement. If no other tables or indexes exist within the affected areas and all relationships are within those areas (and were unloaded), format the area before issuing the DROP and DELETE statements. Be sure to vary the areas offline to the DC/UCF system before formatting them.

5. Redefine the table making any necessary changes.

6. Redefine the indexes and CALC key on the target table.

7. Redefine the referential constraints in which the target table participates. If any of the constraints involve non-empty tables, those constraints must be defined as unlinked.

8. Reload the tables using the LOAD utility statement and the sequential file as input.

   Note: For more information about how to perform the load operation, see Chapter 23, "Loading an SQL-Defined Database".

9. Complete the process as follows:

   - Recreate the views whose syntax was previously saved; examine those view definitions to see if changes are required
   - Respecify privilege definitions on the target table and its referencing views if access is controlled through CA IDMS internal security

Guaranteeing the Integrity of the Data

You must ensure that no updates are made to any of the unloaded tables once their data has been extracted. To *ensure* that no changes are made between the time the data is extracted and the time the tables have been reloaded:

- Prohibit access to the tables by explicitly dropping all views that reference it. This is effective only if all update access to the table is done through a view.
- Revoke all INSERT, UPDATE, and DELETE privileges from the tables (and any matching wildcarded table names) if access is controlled though CA IDMS internal security.

Maintaining Routines and Their Keys

A routine is either an SQL procedure, table procedure or function. This section describes how to:

- Drop a routine
- Change a routine definition and associate or disassociate keys

Dropping a Routine

DROP Routine Statements

To drop a routine, use one of the following statements:

- DROP PROCEDURE to drop an SQL procedure
- DROP TABLE PROCEDURE to drop a table procedure
- DROP FUNCTION to drop an SQL scalar function defined by a CREATE FUNCtion statement

No CASCADE

When you drop a routine (without CASCADE), the following definitions are removed from the dictionary:

- The procedure, table procedure or function that is the target of the drop
- All keys defined on the target routine
- All privileges granted on the routine

With CASCADE

If you specify CASCADE, these additional definitions are removed from the dictionary:

- All views in which the routine is referenced and all views referencing those views.
- All privileges granted on views dropped as a result of cascade processing

Example

In the following example, the table procedure WORK_SCHEDULE is dropped. This also drops any views derived from this table procedure.

```
drop table procedure work_schedule cascade;
```

Modifying a Routine

Types of Changes You Can Make

The following attributes of a routine can be changed:

- External name of the routine
- Estimated rows and I/O counts
- Work area usage, size and attributes
- Mode in which the routine executes
- Definition timestamp
- Default database in effect when the routine is invoked
- Whether the routine should share the encompassing SQL session's transaction
- Programming language that the routine is written in

Additionally, you can also add or remove a key from a procedure or table procedure. Keys are used to assist CA IDMS/DB in calculating the amount of resources that will be consumed by a given routine invocation.

Statements That Modify Routines

To modify a routine's attributes, use one of the following statements:

- ALTER PROCEDURE to alter an SQL procedure
- ALTER TABLE PROCEDURE to alter a table procedure
- ALTER FUNCTION to alter an SQL scalar function defined by a CREATE FUNCTION statement

To associate a new key with a procedure or table procedure, use the CREATE KEY statement. To drop a key, use a DROP KEY statement.

Examples

In the following example, function CORP_DATE is altered to change the external name of the function.

```
alter function corp_date external name corpdate:
```

Chapter 31: Modifying Indexes, CALC Keys, and Referential Constraints

This section contains the following topics:

Overview

This chapter describes methods for creating, dropping, and changing indexes, CALC keys, and referential constraints.

Note: For more information about the SQL DDL statements used in the procedures in this chapter, see the *CA IDMS SQL Programming Guide*.

Maintaining Indexes

This section describes how to:

- Create or drop an index
- Change index characteristics
- Move an index from one area to another

Creating an Index

To create a new index on a column or columns in a table, use the SQL DDL CREATE INDEX statement. If the index is going to map to a new area, see Defining Segments, Files, and Areas for information about defining an area.

Considerations

- If you specify that the index is unique, and data in the key columns is *not* unique, you will receive an error and the index will not be created.
- Each index implies additional runtime processing to handle INSERT, UPDATE, and DELETE statements for the index itself.

Note: For more information about designing indexes, see the *Database Design Guide*.

Example

In the following example, an index is built on the LAST_NAME column in the BENEFITS table.

```
create index be_lname (last_name) on emp.benefits;
```

Dropping an Index

To drop an index from an existing table, use the SQL DDL DROP INDEX statement.

Considerations

A unique index or CALC key is required on all referenced columns in a constraint, and an index including the referencing (foreign key) columns or a CALC key on all referencing columns must exist to support unlinked constraints. If dropping an index would violate either of these rules, the DROP will not be allowed.

Example

In the following example, an optional index is dropped from a table:

```
drop index be_lname from emp.benefits;
```

Changing Index Characteristics/Moving an Index

To modify index characteristics or to move an index from one area to another, use the SQL DDL ALTER INDEX statement.

Types of Changes You Can Make:

The following attributes of a referential constraint can be changed:

- Index block specification
- Index uniqueness
- The area in which an index resides

Statements That Modify Indexes:

To modify index attributes, use the ALTER INDEX statement.

Considerations

- If changing index tuning options, remember to observe referential constraint rules.

Example: Index altered on the BENEFITS table

```
alter index emp_lname (last_name) on emp.benefits
      displacement is 40 pages
           index block contains 30 keys
      in area emp.emp1;
```

Maintaining CALC Keys

This section describes how to:

- Create a CALC key
- Drop a CALC key

Creating a CALC Key

To create a CALC key for an empty table, use the SQL DDL CREATE CALC statement.

If the table is not empty, you must drop and recreate the table, adding the CALC key before reloading the table's data.

Note: For the steps involved in this process, see Chapter 30, “Modifying Schema, View, Table, and Routine Definitions”.

Considerations

Only one location mode is permitted for a table. If the table is stored clustered on an index or constraint, you must drop the clustering index or constraint and re-add it as non-clustered before you can create a CALC key.

Example

In the following example, a unique CALC key is created for the EMPLOYEE table.

```
create unique calc key on emp.employee (emp_id);
```

Dropping a CALC Key

To drop a CALC key from an empty table, use the SQL DDL DROP CALC statement.

If the table is not empty, you must drop and recreate it.

Note: For the steps involved in this process, see Chapter 30, “Modifying Schema, View, Table, and Routine Definitions".

Considerations

You cannot drop a CALC key that is required for implementation of a referential constraint if no index exists to support it. If necessary, either drop the constraint or create an index to support it before dropping the CALC key.

Example

In the following example, the CALC key is dropped from the EMPLOYEE table.

```
drop calc key from emp.employee;
```

Maintaining Referential Constraints

This section describes how to:

- Create or drop linked or unlinked referential constraints
- Modify the tuning characteristics of referential constraints

Creating a Referential Constraint

To create an *unlinked* or *linked* referential constraint, use the SQL DDL CREATE CONSTRAINT statement. CA IDMS/DB checks and rejects any invalid CREATE CONSTRAINT statements.

Considerations

- To create a *linked* constraint if both tables are not empty, you must drop and recreate the tables, defining the linked constraint before reloading the data.

  Note: For steps and considerations involved with this process, see Chapter 30, “Modifying Schema, View, Table, and Routine Definitions.

- When adding an *unlinked constraint* on a non-empty table, CA IDMS/DB ensures that all rows of the table satisfy the constraint. If one or more rows do not satisfy the constraint, the create will not be allowed.

Example

In the following example, a linked referential constraint has been created to make sure that the employee ID in the benefits table is a valid ID by checking it against the employee IDs in the employee table. The referential constraint is indexed and ordered by the fiscal year.

```
create constraint emp_benefits
   benefits (emp_id)
   references employee (emp_id)
   linked index
     order by (fiscal_year desc);
```

Dropping a Referential Constraint

To drop an *unlinked* referential constraint, or a *linked* referential constraint, use the SQL DDL DROP CONSTRAINT statement.

Considerations

If you drop a clustered constraint, the location mode of the referencing table will change as follows:

- If a default index exists, CA IDMS/DB will use it as the clustering index.
- Otherwise, it uses a direct location mode which means that all new rows will be stored in the first page containing enough space to hold the row.

Example

In the following example, the EMP_BENEFITS constraint is removed from the BENEFITS table:

```
drop constraint emp_benefits from benefits;
```

Modifying Referential Constraint Tuning Characteristics

To modify referential constraint tuning characteristics (for example, changing from unlinked to linked or adding an ORDER BY option) use the SQL DDL DROP CONSTRAINT statement, then re-add the constraint using the SQL DDL CREATE CONSTRAINT statement. Certain referential constraint characteristics can be changed with the SQL DDL ALTER CONSTRAINT statement.

For more information about SQL DDL statements see the *SQL Reference Guide*.

Using ALTER CONSTRAINT

Types of Changes You Can Make:

The following attributes of a referential constraint can be changed:

- Index block specification
- Index uniqueness

Statements That Modify Constraints

To modify referential constraint attributes, use the ALTER CONSTRAINT statement.

Considerations

All considerations for modifying a referential constraint apply.

Example: Indexed constraint characteristics are changed

```
alter constraint dept_empl on demo.empl
      displacement is 50 pages
      index block contains 10 keys;
```

Using DROP/CREATE CONSTRAINT

Considerations

All considerations for dropping and creating a referential constraint apply.

Example: Linked referential constraint has been changed to unlinked

```
drop constraint emp_benefits from benefits;
create constraint emp_benefits
       benefits (emp_id)
       references employee (emp_id);
```

Chapter 32: Modifying Non-SQL Defined Databases

This section contains the following topics:

Types of Modifications

Modification of a non SQL-defined database involves modifying any of the components you defined earlier for the schema or subschema. This includes:

- Adding or deleting schemas
- Adding, modifying, or deleting schema areas
- Adding, modifying, or deleting schema records
- Adding, modifying, or dropping indexes and sets

Note: For more information about modifying physical definitions, see Chapter 27, “Modifying Physical Database Definitions".

Changes to Schemas and Subschemas

In general, when you change a database, you must modify the schema code and revalidate the schema. However, changing the schema has an impact on other components of the CA IDMS/DB environment. If you add or delete an area from a schema, you may have to add or delete that area in one or more segments and regenerate DMCLs. You will also have to modify and recompile some or all subschema definitions compiled under the original schema to reflect changes made to the schema.

If you access the non-SQL defined data through SQL, you may also need to recompile access modules and drop and recreate SQL view definitions.

The primary tool for changing a schema is the schema compiler.

Steps to Modify the Schema

The steps to make any schema modification are as follows:

1. Change and re-validate the necessary schema and subschema definitions
2. Change the actual *data* (if it exists) to fit the new database specifications using the RESTRUCTURE, MAINTAIN INDEX, REORG or UNLOAD/RELOAD utility statements
3. Revise and recompile any application programs that may have been affected by the above changes
4. Test to ensure that the change has been made correctly.

Changes to Subschemas

Subschemas identify selected areas, records, elements, and sets of the database. They also define logical records and establish security by restricting runtime access to the database.

Any time you make a change to any of the above components in your CA IDMS/DB environment, you will have to change one or more of your subschemas.

The primary tool for changing subschemas is the subschema compiler.

Methods for Modifying

Depending on the type of change you want to make to a non-SQL defined database, you would do one of the following:

- Change the definition
- Change the definition and additionally use one or more utility statements

Basic Definition Change

To change a logical database definition when there is no impact on data, you can use the schema compiler (or another compiler). This type of change takes effect without requiring the use of a utility statement.

An example of a change in which there is no data impact is adding a new area to a schema.

Definition Change Using Utility Statements

For database changes that have an impact on data, you must change the database definition and additionally use an appropriate utility statement:

- **RESTRUCTURE**—Modifies record occurrences to fit new schema specifications. RESTRUCTURE allows you to:
  - Insert new data items anywhere in a record
  - Delete existing data items
  - Change the length and position of data items
  - Change the format of a record from fixed length to variable length or from variable length to fixed length
  - Add or remove record compression
  - Delete chained sets and add or delete set pointers
- **REORG and UNLOAD/RELOAD**—Reorganizes data when changes are made to the placement of records and indexes within the database (for example, moving a record from one area to another).
- **MAINTAIN INDEX**—Builds, rebuilds, or deletes indexes in the database. You use this utility whenever you need to make a structural change to the database involving indexes (for example, adding a new index to the database).

Procedure for Modifying the Non-SQL Definitions

Step 1: Copy the Original Schema and Global Subschema

1. Create a new schema which is identical to the *original* schema.
2. Create a global subschema for the new schema with a name which is different from that of any other subschema in the dictionary. Include in the subschema all areas, records, and sets associated with the schema.

Step 2: Modify the New Schema and Subschema

1. Make the necessary changes to the *new* schema definition.
2. Validate the schema.
3. Regenerate the global subschema, modifying it if necessary.

Step 3: Modify the Segment and DMCL, If Necessary

Note: You need to modify segments and DMCLs only if you add or remove an area or make other changes to the physical definition in addition to changing the schema.

1. Make the appropriate changes in the segment definition. Make sure that subareas and other symbolics are defined appropriately.
2. Generate, punch, and link all DMCLs containing the altered segment.

Step 4: Make Changes to the Data

Note: Not all schema changes require data changes. See Chapter 33, "Modifying Schema Entities" for the steps needed in each case.

1. Backup the area or files.
2. Use the appropriate utility or user-written program to change the data.
3. Verify the change using IDMSDBAN and/or a retrieval program, CA OLQ, or CA Culprit.
4. Backup the altered areas or files.

Step 5: Complete the Change

1. Update the *original* schema in the same way that the copy was changed.
2. Regenerate all subschemas associated with the original schema that are affected by the change, modifying them if necessary to add new areas, records, or sets.
3. Recompile all access modules affected by the change, using the ALTER ACCESS MODULE statement with the REPLACE ALL option.
4. Drop and recreate all SQL views affected by the change.
5. Make the new subschemas, DMCLs, and files available to your runtime environment.

Considerations

The procedure outlined above requires that changes first be made to a copy of the original schema and only after all other steps have been completed are the changes made to the original schema. This approach ensures that the original schema continues to describe the data until the altered areas are made available to the runtime environment. You should use this (or a similar approach) if during the process:

- CA OLQ, CA Culprit, or dynamic SQL users will be accessing the original schema definition
- Application programs will be compiled against the original schema and must access the data before it has been changed.

If the above is not a concern or if no data changes are necessary, then the initial modifications can be made to the *original* schema rather than a copy, avoiding the necessity of replicating those changes later.

RESTRUCTURE Utility Statement

What RESTRUCTURE Does

The RESTRUCTURE utility statement modifies record occurrences to fit new schema specifications. You run RESTRUCTURE in local mode using a subschema associated with a schema that describes the database *before* restructuring.

RESTRUCTURE does not require that the database be unloaded and reloaded. Database keys remain unchanged. New database procedures can be executed during restructuring. For example, IDMSCOMP can be executed to compress previously uncompressed records.

Steps for RESTRUCTURE

To make changes using RESTRUCTURE, follow the procedure described in the section “Procedure for Modifying the Non-SQL Definitions”, except add the steps listed in the following table.

After ...	Do this
Modifying the schema and subschemas	Execute the schema compare utility (IDMSRSTC) to generate IDMSRSTT macro statements for use in the database restructure
Executing IDMSRSTC	Assemble the IDMSRSTT statements into a base restructuring table and use the table with the RESTRUCTURE utility statement; use a subschema that describes the database *before* restructuring

After ...	Do this
Executing RESTRUCTURE	Connect any new pointers to existing sets using the RESTRUCTURE CONNECT utility statement; use a subschema that describes the database *after* restructuring
Executing RESTRUCTURE CONNECT	Write a program to connect pointers in new sets to existing records

Note: For more information about IDMSRSTC, RESTRUCTURE, and RESTRUCTURE CONNECT, see the *Utilities Guide*.

REORG and UNLOAD/RELOAD Utility Statements

What REORG and UNLOAD/RELOAD Do

The REORG and UNLOAD/RELOAD utility statements reorganize databases by unloading existing records and reloading them into another database. To reorganize a database, you follow one of two approaches:

- Use the UNLOAD utility statement to offload data to an intermediate file. Use the RELOAD utility statement to store the record data into another database, build index structures, and connect related records together in set structures.
- Use the REORG utility statement to both offload and reload data.

The choice of which approach to use depends on a number of factors:

- The operating system in which the utility is to execute. Currently, REORG is only supported in z/OS.
- The size of the database and the amount of time that you have available to reorganize it. REORG uses parallel processing to reduce the elapsed time that it takes to reorganize a database.
- The types of changes being made. The REORG utility supports more types of changes and in certain cases makes implementing those changes easier. For example, the REORG utility allows you to:
  - Change the CALC key of a record without unloading and reloading twice
  - Change the fields comprising an index key
  - Change the order of an index

  Note: For more information about the types of changes that can be made through the two utilities, see the *CA IDMS Utilities Guide*.

Steps for unloading and reloading a database

To make changes using REORG or UNLOAD/RELOAD, follow the procedure described in 32.2.2, "Procedure for Modifying the Non-SQL Definitions", adding the steps listed in the following table.

After ...	Do this
Modifying the schema and subschemas	Use the REORG or UNLOAD/RELOAD statements to reorganize the data using subschemas that reflect both the old and new schema definitions.
Unloading the data	Use the FORMAT utility statement to initialize the files into which the data will be reloaded.

Note: For more information about REORG, UNLOAD and RELOAD, see the *CA IDMS Utilities Guide*.

MAINTAIN INDEX Utility Statement

What MAINTAIN INDEX Does

The MAINTAIN INDEX utility statement allows you to build, rebuild, and delete both system-owned and user-owned indexes (indexed sets). You can also change the characteristics of an index, such as changing an index key from a compressed to an uncompressed format.

Steps to Modify Indexes

To make changes to an index, follow the procedure described in 32.2.2, "Procedure for Modifying the Non-SQL Definitions", adding the steps listed in the following table.

After ...	Do this
Modifying the schema and global subschema	**For system-owned indexes:** Use MAINTAIN INDEX to build, rebuild, or delete an index. **For user-owned indexes (indexed sets):** Write a program that calls IDMSTBLU and passes descriptor information

Note: Depending on the operation, you will need either a subschema reflecting the old schema, the new schema, or both.

Note: For more information about MAINTAIN INDEX and IDMSTBLU, see the *CA IDMS Utilities Guide*.

Chapter 33: Modifying Schema Entities

This section contains the following topics:

Overview

This chapter describes:

- The procedure to modify a schema or schema entities when the database is empty.
- Specific procedures to add, delete, or modify schema or schema entity definitions when the database is *not* empty. These procedures include only those that require a utility to effect the change.

Modifications to an Unloaded Database

What components are affected

Changes to a schema or schema entity in an unloaded database affects:

- The schema
- The subschemas that reference the schema
- The access modules that reference the schema
- SQL views that reference the schema

Steps

1. Modify the schema and any schema entities, as desired
2. Validate the schema
3. Regenerate any affected subschemas
4. Drop and recreate SQL views that reference the schema, as necessary
5. Alter affected access modules using the REPLACE ALL option

Schema Modifications

This section describes how to:

- Delete a schema
- Change schema characteristics

Deleting a Schema

What Components are Affected

When you delete a schema, the definitions of the schema and all subschemas associated with the schema are removed from the dictionary.

Steps to delete a schema

To delete a schema from the dictionary:

1. Delete the schema
2. Delete load modules associated with the deleted subschemas
3. Delete files that contain the data
4. Delete the segment(s) corresponding to the schema
5. Regenerate all affected DMCLs
6. Remove the segment(s) from the database name table
7. Delete SQL schema(s) referencing the non-SQL schema

Considerations

When you delete a schema, subschemas associated with that schema are also deleted. The subschema load modules are not deleted.

In addition, the physical database definition(s) that apply to the schema's areas are not automatically deleted. You must modify the physical database definitions to delete the areas and regenerate all affected DMCLs.

Changing Schema Characteristics

Schema characteristics include:

- Description
- Memo date
- Assignment rules for record IDs
- Security specifications
- User-defined information (class/attribute and user-defined comments)

What components are affected

When you modify characteristics of a schema, only the schema definition is affected. These characteristics do not impact critical definitions within the schema or its subschemas, so a VALIDATE statement is not required.

Area Modifications

Types of changes

This section describes how to make the following area-related changes:

- Add or delete an area in an existing schema definition
- Add, remove, or change procedures associated with the area

Adding or Deleting an Area

What Components are Affected

Adding or deleting an area in the schema affects the schema and subschemas referencing the area to be deleted, segments describing related physical databases, and DMCLs in which those segments are included.

Steps to add an area

1. Modify the schema
2. Add the new area
3. Define one or more records or system-owned indexes associated with the area
4. Validate the schema
5. Add the new area to one or more subschemas
6. Format the new area using the FILE option of the FORMAT utility statement

Steps to delete an area

1. Modify the schema
2. Modify existing records mapping to the area to be deleted so that they map to a different area
3. Delete the area
4. Validate the schema
5. Regenerate any affected subschemas

Considerations

- If existing records are to reside in a new area, see 33.5.6, "Changing a Record's Area".
- If an existing index is to reside in a new area, see 33.7.2, "Changing the Location of an Index".
- After you have added an area to a schema or deleted an area from a schema, make sure you update the DMCL module appropriately.

Changing Area Characteristics

What components are affected

When you add or delete area procedures, the area and schema definitions are affected. All subschemas which include the area must be regenerated and all access modules accessing a record in the area must be altered.

Steps to change area characteristics

See 33.2, "Modifications to an Unloaded Database" for the steps to modify an empty database.

Considerations

Remember to respecify all database procedures in the order that they are to be called when you add, remove, or change the procedures associated with an area.

Record Modifications

Types of changes

This section describes the following record-related changes:

- Adding or deleting a record in your schema
- Changing a record's CALC key
- Changing a record's location mode
- Changing a record's area
- Changing a record element
- Changing record procedures

Adding Schema Records

What components are affected

Adding schema records affects the schema.

Steps to add a record

1. Add the record using DDDL statements
2. Modify the schema
3. Add the record to the schema using SHARE STRUCTURE
4. Validate the schema
5. Modify any subschemas that should contain the new record
6. Add the new record to each subschema
7. Regenerate the subschemas

Considerations

If the record participates in a set with existing records, you must use the RESTRUCTURE utility statement to add pointer positions to the existing records. You must also write a program that connects the records into proper set occurrences.

Note: For more information about adding a set to a schema, see the section "Adding or Deleting a Set".

Deleting Schema Records

What components are affected

Deleting schema records affects the schema *and* the data. It also affects subschemas and access modules that reference the record and any other records connected to the record through sets. SQL views referencing the record become invalid.

Steps to delete a record

To delete a record from the schema where data has been loaded:

1. Write and execute a program to erase all occurrences of the record
2. Create a new schema based on the original schema omitting the record and omitting any affected sets
3. Validate the schema
4. Use the schema compare utility (IDMSRSTC) to generate the IDMSRSTT macro statements
5. Restructure the database using the RESTRUCTURE utility statement

   Note: If the record does not participate in any set relationships, there is no need to restructure the database.
6. Complete the process by updating the original schema definition, regenerating subschemas, altering affected access modules, and dropping affected views

Considerations

- If you created the record using DDDL statements, the record definition will remain in the dictionary after it has been deleted from the schema
- If you created the record using schema DDL and the record has *not* been copied into any other schema, its definition will be deleted from the dictionary after it has been deleted from the schema definition.
- If the record participates in a set relationship, you have to remove the set from the schema definition or modify the definition before validation
- Regenerating affected subschemas will remove the record from the subschema definition
- When you erase occurrences of the record, you may have to use a subschema derived from a schema where the sets in which the record is an owner have been changed to optional. This permits the member record to be disconnected from the owner record rather than being erased.

Changing a Record's CALC Key

Types of Changes

You can make the following changes to a record's CALC key:

- Replace one or more elements in the CALC key
- Add or remove elements in the CALC key
- Change the picture or usage of an element in the CALC key

What components are affected

Both the schema record definition and the data are affected. Subschemas and access modules that reference the record are also affected.

Steps to change the CALC key

To change the CALC key of a schema record where data has been loaded:

1. Add a new schema based on the original schema
2. Modify the record in the new schema specifying the new CALC key or any changes to fields currently participating in the record's CALC key
3. Validate the schema
4. Execute the RESTRUCTURE utility if any fields within the record's CALC key have had their definitions changed or any of the conditions discussed under "Considerations" apply
5. Create a new global subschema if applicable
6. Unload and reload the database
7. Complete the process by updating the original schema definition, regenerating subschemas, and altering affected access modules

Considerations

- If a field to be added to the CALC key does not exist in the record, add the field using RESTRUCTURE and initialize it before unloading and reloading the data. Initialize the field using restructure or a user-written program.

  Note: For information about adding a new record element, see 33.5.7, "Modifying Record Elements".

- If using UNLOAD and RELOAD (instead of REORG) to change a CALC key, the following additional considerations apply:

  - If the control length of the record is changing as a result of the change to the CALC key and the record is compressed or variable length, you must do one of the following:

    - Use RESTRUCTURE to alter the control length before unloading and reloading the data.

    - Unload and reload the data twice (using the old subschema on the first unload and the new subschema on the second)

  - UNLOAD/RELOAD clusters VIA records based on the CALC key defined in the subschema used to *unload* the data. Therefore, you need to do a second UNLOAD/RELOAD to properly cluster VIA records if the subschema used to unload the data describes the old CALC key.

    As an alternative, you can use the new subschema for unloading. This ensures that the new CALC key is used to determine target pages for both the CALC record and its associated VIA records. However, the new subschema can be used to unload data only if no other changes have been made to the record (such as the record's area, set pointers, etc.).

    In some cases, multiple changes can be accommodated by using an intermediate schema/subschema to unload the data. For example, to change the CALC key of a record and also move it to a new area, unload the data using a subschema that describes the record's new CALC key but old area. Reload the data using a subschema describing the new CALC key and the new area.

Changing the DUPLICATES Option on a CALC or SORT Key

Types of Changes

You can make the following changes to the DUPLICATES option on a record's CALC or sort key:

- Duplicates first to duplicates last
- Duplicates last to duplicates first
- Duplicates not allowed to duplicates first/last
- Duplicates first/last to duplicates not allowed

What components are affected

The schema definition is affected. Depending on the change, the data may also be affected. All subschemas and access modules referencing the record are also affected.

Steps to change the duplicates option

See 33.2, "Modifications to an Unloaded Database" at the beginning of this chapter for the steps to change the duplicates option from:

- Duplicates first/last to duplicates not allowed
- Duplicates not allowed to duplicates first/last

To change the duplicates option from first to last or from last to first:

1. Write a program using a subschema that specifies duplicates first for the CALC or sort key. The program must
   - Modify the CALC or sort key value to a dummy value
   - Modify the CALC or sort key value to its original value
   - Read each record that has duplicate values, using either OBTAIN CALC DUPLICATE or OBTAIN NEXT IN SET to retrieve duplicate records in the current order

   This has the effect of reversing the order of the duplicate records.
2. Modify the schema
3. Modify the record changing the duplicates option
4. Validate the schema
5. Regenerate any affected subschemas
6. Alter affected access modules using the REPLACE ALL option

Considerations

- When you change from duplicates first or last to duplicates not allowed, make sure that no duplicate key values exist in the database.
- When changing from duplicates first to last or last to first, write a conversion program to logically reorder the record occurrences in the database.

  Using the approach described above, the program must execute using a subschema specifying duplicates *first*. Therefore, it should use a subschema created either before or after the schema has been changed depending on whether the duplicates option is being changed from or to duplicates first.

Changing the Location Mode of a Record

Types of Changes

These are the possible location mode changes for a record in the database:

- CALC to VIA
- CALC to DIRECT
- DIRECT to CALC
- DIRECT to VIA
- VIA to CALC
- VIA to DIRECT
- VIA one set in the schema to VIA another set

What components are affected

The record definition and the data are affected. Subschemas and access modules referencing the record are also affected.

Steps to change the location mode

To change the location mode of a schema record where data has been loaded:

1. Add a new schema based on the original schema
2. Modify the record in the new schema to specify the new location mode
3. Validate the schema
4. Create a new global subschema
5. Unload and reload the database
6. Complete the process by updating the original schema, regenerating affected subschemas, and altering affected access modules.

Considerations

- If the storage mode is being changed to VIA:
  - If the set does not exist, take the following actions before unloading and reloading the data:
    - Add the set to the schema and subschemas
    - Write a program to connect the members to the appropriate owner occurrence
  - Unload and reload the data using either the REORG or the UNLOAD/RELOAD utilities.
  - If the REORG utility is used, the unload phase takes additional time to retrieve the member records because it does so by walking the new (non-clustering) VIA set
  - If UNLOAD/RELOAD are used, you must do one of the following:
    - Either unload and reload the data a second time in order to cluster the member records correctly
    - Use an intermediate subschema to unload the data. The intermediate subschema must describe the original database except that the member record is VIA the new set. Using this technique results in longer unload times because the member records are retrieved by walking the new (non-clustering) VIA set.
- If the storage mode is being changed to CALC, see 33.5.3, "Changing a Record's CALC Key" for considerations on doing this.
- If the storage mode is being changed to DIRECT:
  - You must unload and reload the data if the original location mode was CALC but not if it was VIA.
  - The record is reloaded into the same page if the page is within the record's target page range. If the old page is not part of the target page range, the record is stored in its target range proportionally to its position in its original range. If this is not acceptable, you may need to write a user-written program to unload the database and the FASTLOAD utility to reload it.

Changing a Record's Area

Types of Changes

You can move a record from one area to another or change the portion of an area in which a record is stored.

Note: If a subarea symbolic is associated with the record, you change the portion of the area in which the record is stored by changing the physical area definition and regenerating DMCLs. See Chapter 27, “Modifying Physical Database Definitions” for more information.

What components are affected

The schema record definition and data area affected. Subschemas and access module that reference the record are also affected.

Steps to change the record's area

To change the area (or portion of an area) in which record occurrences are stored when data has been loaded:

1. Create a new schema based on the original schema
2. Add the area, if necessary, to the new schema
3. Modify the record in the new schema to specify the new area or subarea/offset
4. Validate the schema
5. Create a new global subschema
6. Unload and reload the database
7. Complete the process by updating the original schema, regenerating affected subschemas, and altering affected access modules.

Considerations

- If you had to add the area to the schema, you must explicitly add it to subschemas associated with the record. You must also explicitly add the area to all applicable physical database definitions and regenerate affected DMCLs.
- If you increase the page range of a record whose location mode is other than CALC, you do not need to unload and reload the data provided the new page range includes all pages of the original range.

Modifying Record Elements

Types of changes

These are changes you can make to an element within a schema record:

- Adding or removing a record element
- Changing the picture or usage mode of an element

What components are affected

The record definition is affected. If data has been loaded, the data may also be affected. Subschemas in which the record is included are affected as are programs compiled from those subschemas. Access modules and SQL views that reference the record are also affected.

Steps to change the record element

To make any of the above changes when data has been loaded:

1. Using DDDL statements, create a record with a new version number and same name having the revised structure
2. Create a new schema based on the original schema with the new record
3. Validate the schema
4. Use the schema compare utility (IDMSRSTC) to generate the IDMSRSTT macro statements
5. Restructure the database
6. Complete the process by updating the original schema, regenerating affected subschemas, altering access modules, and dropping and recreating affected SQL views

Considerations

- You can replace filler elements with record elements whose total length equals that of the filler element without creating a new version of the record. The new elements are immediately reflected in the schema. The next time any programs that use that schema record are compiled, the new elements appear. Affected subschemas are flagged for regeneration.
- You should initialize any 'filler' fields or fields whose picture or usage has been changed using either RESTRUCTURE or a user-written program.
- A record element in a schema-owned record can be replaced with elements of the same name

- If you want to maintain consistency among the record version numbers in your schema:

    1. Complete all of the steps above
    2. Delete the original version of the record
    3. Modify the record using DDDL statements to change its version number

    You do not have to modify the schema.

- Non-structural changes can be made directly to schema-owned records using DDDL. For example, you can change the external picture of a record element even if it is associated with a schema.

    Note: For more information about the types of changes that can be made to schema-owned records, see the *CA IDMS IDD DDDL Reference Guide*.

- If you change the format (picture or usage) of an element used in a CALC or sort key, additional steps may be needed to convert the data.

Changing Other Record Characteristics

Types of Changes

You can make the following changes to the characteristics of a record (changes other than those described previously in this chapter):

- Record ID
- Record synonyms
- VSAM type
- Minimum root and minimum fragment length
- Whether the record is compressed or uncompressed

Note: For information about dropping or adding a database procedure associated with a record, see 33.5.9, “Adding and Dropping Database Procedures”.

What components are affected

The record definition is affected and the data is affected if changing the record ID or compression. All subschemas and access modules that reference the record are affected. SQL views are affected only if the SQL synonym for an element is changed.

Steps to make the change

To modify the VSAM type, record synonyms, or minimum root and fragment lengths, follow the procedure described in Modifications to an Unloaded Database at the beginning of this chapter.

Considerations

- To change the record ID when data exists, write a program that offloads and reloads that data.
- Optionally, unload and reload the data to reorganize existing data after changing the minimum root or minimum fragment.
- To change a record from compressed to uncompressed, you must either unload and reload the data or use RESTRUCTURE to alter the data.
- If changes to the record elements do not affect control fields, all you need to do is issue a DDDL MODIFY RECORD statement.

  Note: For more information about modifying non-IDD owned records, see the *CA IDMS IDD DDDL Reference Guide*.
- If you change the SQL synonym for one or more elements, then you must drop and recreate all SQL views that reference the record. You must also change all programs that refer to those elements in an SQL statement.
- If you change the VSAM type, ensure that appropriate changes, if necessary, are made to the VSAM definition using the IDCAMS utility.

Adding and Dropping Database Procedures

What components are affected

If you implement a new database procedure, or change the name of an existing procedure, it will affect the schema and one or more subschemas. It *may* also require that you restructure the database, if the purpose of the procedure is to alter the physical data (for example, record compression). All subschemas and access modules that reference the record are also affected by procedure changes.

Steps to make the change

To add, modify, or delete database procedures that have no effect on the data, follow the procedure described in 33.2, "Modifications to an Unloaded Database".

Considerations

- If a new database procedure *does* affect the data, write and compile the new procedure and then use the RESTRUCTURE utility statement to change the existing data by specifying the new procedure in a NUPROCS macro.

  If database procedures are already associated with the record, they may need to be removed from the schema and subschema before executing RESTRUCTURE. The existing procedures, if invoked, will be called *after* all NUPROCS procedures have been called. If, for example, the new procedure compresses the data in the record, the existing procedures may not work properly. To overcome this problem, either execute RESTRUCTURE using a subschema derived from an intermediate schema in which all procedures normally called before the new procedure have been removed from the record or unload and reload the data to add the new procedure.

- You can add or remove procedures that affect the data by unloading and reloading it. To do this, create a new schema and subschema containing the revised procedure calls. Unload the data using the old subschema and reload it using the new subschema.

- To change a database procedure for an area, all calls must be respecified.

Set Modifications

Types of changes

The following set-related changes can be made:

- Add or remove a set
- Change the mode (index or chain)
- Add or remove set pointers
- Change set order
- Change membership options

Adding or Deleting a Set

What components are affected

The schema set definition and data are affected. Segments and DMCLs may also be affected if a set is indexes and a symbolic index specification needs to be added, removed, or replaced in the physical definition. Subschemas and access modules that reference either the owner or member of the set are also affected.

Steps to add or delete a set

To add or delete a set when data has been loaded:

1. Create a new schema based on the original schema but containing the new set or omitting the deleted set
2. Validate the schema
3. Create a global subschema for the new schema
4. Use the schema compare utility (IDMSRSTC) to generate the IDMSRSTT macro statements
5. Restructure the database using the RESTRUCTURE utility statement
6. If adding a set, write a program to connect member record occurrences to the appropriate owner occurrences.
7. Complete the change by updating the original schema, regenerating affected subschemas, and altering affected access modules.

Considerations

- Both records participating in a new set must be defined to the schema
- If you replace an existing set with a new set, do not use the AUTO parameter; specify the actual pointer positions. This eliminates the possibility that the schema compiler will identify different pointer positions than exist in the loaded database.
- When deleting an existing set from a schema and a participating record contains pointer positions for sets *beyond* the deleted set's pointer positions, you must renumber the remaining positions. You cannot leave unused pointer positions.
- If you delete a set, the set is also deleted from all subschema descriptions.
- If you delete the owner record within a set, the set is automatically deleted and both the set and deleted record are removed from all subschema descriptions.
- If you delete the member record within a set, the set remains. You receive an error on validation if there are no remaining members in the set (as in a multimember set)
- If you want a new set to be included in a subschema, you must modify the subschema and add the set to the subschema.
- Regenerating affected subschemas will remove a deleted set from all subschemas.
- When you delete a set, alter and recompile all programs that use the set

Changing Set Mode

Types of changes

You can change the mode of a set from chain to index or vice versa.

What components are affected

The schema set definition and data are affected. All subschemas and access modules that reference either the owner or member records are also affected.

Steps to change from chained to indexed

To change a chained set to an indexed set when data has been loaded:

1. Create a new schema based on the original schema
2. Modify the set in the new schema to change the set mode
3. Validate the schema
4. Create a global subschema
5. Write a program that sweeps the area, walks each set, and calls IDMSTBLU to perform a BUILD function.
6. Restructure the database if needed to remove old pointer positions and add new ones.
7. Execute MAINTAIN INDEX from SORT3 using the output from step 5 as input
8. Complete the change by updating the original schema, regenerating affected subschemas, and altering affected access modules.

Steps to change from indexed to chained

To change an indexed set to a chained set when data has been loaded:

1. Create a new schema based on the original schema
2. Modify the set in the new schema to change the set mode
3. Validate the schema
4. Create a global subschema
5. Write a program that sweeps the area and calls IDMSTBLU to perform a DELETE function and also produces a work file for input to step 8.

6. Use the output generated by IDMSTBLU as input to MAINTAIN INDEX and run it from SORT3 to delete each index occurrence

7. Restructure the database as needed to remove old pointers positions and add new ones

8. Sort the workfile produced by IDMSTBLU by owner key, member symbolic key, or set position.

9. Write a program to:

   a. Read the sorted output

   b. Obtain owner by db-key

   c. Obtain member by db-key

   d. Connect the member to the set

10. Complete the change by updating the original schema, regenerating affected subschemas, and altering affected access modules.

Considerations for the change from indexed to chained

- When you submit the RESTRUCTURE utility statement to initialize pointers (and possibly to delete pointers), you must initialize all existing pointer positions in the owner and member records that will be *re-used* for the chained set. If this is not done, you will be unable to connect the members to their owners in Step 9.

- The work file produced in Step 5 should contain the following information:

  - The dbkey of each owner record occurrence
  - The dbkey of each member record occurrence
  - The position of each member record with the set (if its necessary to maintain the same set order)
  - The sort key of the member record within the set (if the set order is changing or the order of duplicates does not have to be maintained)

Adding and Dropping Set Pointers

Types of changes

You can make the following changes to set pointers:

- Add or remove prior or owner pointers from a chain set
- Add or remove owner pointers from an indexed set

What components are affected

When you change the prior or owner pointers defined to a set, the schema set definition and data are affected. Subschemas and access modules that reference either the owner or member records are also affected.

Steps to add or drop set pointers

To add or drop set pointers when data has been loaded:

1. Create a new schema based on the original schema but containing the modified set pointers
2. Validate the schema
3. Create a global subschema
4. Use the schema compare utility (IDMSRSTC) to generate the IDMSRSTT macro statements
5. Restructure the database using RESTRUCTURE
6. If you add a prior or owner pointer to an existing set, fill in the pointer values using RESTRUCTURE CONNECT
7. Complete the process by modifying the original schema, regenerating affected subschemas, and altering affected access modules.

Considerations

- When adding or deleting pointers, do not use the AUTO parameter; specify the actual pointer positions. This eliminates the possibility the schema compiler will identify different pointer positions than exist in the loaded database.
- When deleting a pointer from a set in a schema and a participating record contains pointer positions *beyond* the deleted pointer, you must renumber the remaining positions. You cannot leave unused pointer positions.

Changing Set Order

Types of changes

You can make the following order-related changes:

- Change from SORTED to unsorted (NEXT, PRIOR, FIRST, LAST) order
- Change from unsorted to sorted
- Change one of the unsorted orders to another
- Change the sort key or collating sequence of a sorted set

What components are affected

When you change NEXT, PRIOR, FIRST, LAST, or SORTED specifications, the schema set definition and data are affected. Subschemas and access modules that reference either the owner or member records area are also affected.

Steps to change set order

Follow the steps listed in 33.2, "Modifications to an Unloaded Database" at the beginning of this chapter if both of the following statements are true:

- You are changing a chained or an unsorted indexed set to NEXT, PRIOR, FIRST, or LAST
- It is not important to re-order existing data

Steps to re-order chain and unsorted indexed sets

To change the set order of a chained or unsorted indexed set, member records must be re-ordered:

1. Create a new schema based on the original
2. Modify the set to change the set order
3. Validate the schema
4. Create a global subschema
5. Write a conversion program that disconnects and re-orders (in the desired sequence) each member record occurrence
6. Complete the process by updating the original schema, regenerating affected subschemas and altering affected access modules

Steps to re-order sorted indexed sets

To change the sort key of a sorted indexed set or to change an indexed set from unsorted to sorted and vice versa, follow the procedure for re-ordering chain sets except replace Step 5 with the following:

1. Write a program that sweeps the area and call IDMSTBLU with a REBUILD function
2. Use the output from the step above as input to MAINTAIN INDEX and run MAINTAIN INDEX from SORT3

Considerations

When you change the set order from or to SORTED or when you change the sort key of a sorted set, the control length of the member record may change. If it does, and the member record is compressed or variable in length, you must use RESTRUCTURE to change the control length of existing record occurrences.

Changing Set Membership Options

Types of changes

You can change a MANDATORY set to OPTIONAL and vice versa. You can also change an AUTOMATIC set to MANUAL.

What components are affected

When you change membership options, the schema set definition is affected. Subschemas and access modules that reference either owner or member record are also affected.

Steps to change membership options

To change membership options, regardless of whether data is loaded, follow the steps outlined in 33.2, "Modifications to an Unloaded Database".

Considerations

Changing membership options may impact existing application programs. Consider the following:

- If you change from AUTOMATIC to MANUAL or vice versa, programs that STORE member records may need to connect records into the set using a CONNECT statement or no longer issue such a CONNECT.
- If you change from OPTIONAL to MANDATORY, programs that DISCONNECT members from the set *must* be changed and programs that ERASE owner records may need to be changed.
- If you change from MANDATORY AUTOMATIC to any other option, programs that obtain the owner of the set may be affected (because a given member occurrence may not have an owner).

Index Modifications

Types of changes

You can make the following types of changes to system-owned indexes:

- Add or remove an index
- Change the area in which an index resides
- Change index characteristics
- Change from linked to unlinked or vice versa

Adding or Deleting System-Owned Indexes

What components are affected

When you add or remove a system-owned index, the schema set definition and the data are affected. All subschemas and access modules that reference the member record are also affected.

Steps to add an index

To add an index when data has been loaded:

1. Add a new schema based on the original schema adding the new index
2. Validate the schema
3. Create a global subschema for the new schema
4. If the index is linked, add the index pointer position to the member record using the schema compare utility (IDMSRSTC) and RESTRUCTURE
5. Build the index structure using the new subschema using the MAINTAIN INDEX utility statement
6. Complete the process by updating the original schema, regenerating affected subschemas, and altering affected access modules

Steps to remove an index

To remove an index when data has been loaded:

1. Add a new schema based on the original schema removing the index
2. Validate the schema
3. Create a global subschema for the new schema
4. Delete the index structure using an old subschema and the MAINTAIN INDEX utility statement
5. If the index is linked, remove the index pointed from the member record using the schema compare utility (IDMSRSTC) and the RESTRUCTURE utility statement
6. Complete the process by updating the original schema, regenerating affected subschemas, and altering affected access modules

Considerations

- The index you delete from the schema will automatically be deleted from any affected subschemas when you request that affected subschemas be regenerated.
- If an index is added or removed, it may change the control length of the record. If it does and the record is compressed or variable in length, you must change the control length of existing data using RESTRUCTURE.

Changing the Location of an Index

Types of changes

You can change the area (or portion of an area) in which the index structure resides.

What components are affected

The schema set definition and data are affected. Subschemas and access modules that reference the member record are also affected.

Note: If a subarea symbolic is associated with the index, you change the portion of the area in which the index is stored by changing the physical area definition and regenerating DMCLs. See Chapter 27, “Modifying Physical Database Definitions” for more information.

Steps to change the area

To change the area (or portion of an area) in which an index resides when data has been loaded:

1. Add a new schema based on the original schema
2. Add the area, if necessary
3. Modify the indexed set to map to the new area or subarea/page range
4. Validate the schema
5. Create a new global subschema
6. Rebuild the index using both an old and new subschema using the MAINTAIN INDEX utility statement

Considerations

- If the area does not exist in the subschema, you will receive an error when you issue REGENERATE AFFECTED SUBSCHEMAS.

Changing Index Characteristics

Types of changes

You can change the following index-related characteristics:

- Key compression
- Number of entries in an SR8 record
- Index displacement
- Index key or collating sequence

What components are affected

The schema set definitions and data are affected. Subschema and access modules that reference the member record are also affected.

Steps to change index characteristics

To change index characteristics when data has been loaded:

1. Add a new schema based on the original schema modifying the set characteristics
2. Validate the schema
3. Create a global subschema
4. Rebuild the index using the MAINTAIN INDEX utility and the REBUILD option
5. Complete the process by updating the original schema, regenerating affected subschemas, and altering affected access modules

Considerations

- If you change the index key, the control length of the member record may change. If it does, and the member record is compressed or variable in length, you must use RESTRUCTURE to change the control length of existing record occurrences.
- When you execute the MAINTAIN INDEX utility statement, use the REBUILD option:
  - If you change key compression, you must specify the name of an *old* subschema in the USING parameter and the name of a *new* subschema in the NEW SUBSCHEMA parameter.
  - If you change the symbolic key or the collating sequence, you must specify the *new* subschema in the USING parameter.

  If both types of change are being made at once, you will need to run MAINTAIN INDEX twice, once to delete the existing index (using the old subschema) and once to build the new index (using the new subschema).

Adding or Deleting Index Pointers

Types of changes

You can delete or add index pointers. The index pointer in a member record is optional for system-owned indexes.

Adding or deleting index pointers

To add or delete an index pointer:

1. Modify the schema specifying INDEX POSITION IS NONE
2. Add or delete the pointer position using the RESTRUCTURE utility statement
3. Rebuild the index using the MAINTAIN INDEX utility statement

Chapter 34: Modifying Subschema Entities

This section contains the following topics:

Overview

Affect on Applications Associated with the Subschema

Changes you make to a subschema impact application programs associated with that subschema. In general, when you add, modify, or delete a subschema entity and regenerate the subschema, follow the appropriate procedure in the following table:

If a program...	You should...
Is associated with the subschema but does not need to access the new entity (area, record, or set)	Not have to recompile the program
Is associated with the subschema and needs access to a new entity or has access to a modified entity	■ Alter the program as needed ■ Recompile the program

Regenerating the Subschema

Before you can use the subschema, you must regenerate it as described in Chapter 15, “Subschema Statements".

If you want to use the subschema before the system is recycled, you must issue a DCMT VARY PROGRAM .. NEW COPY command. This statement causes the regenerated subschema to be loaded into the program pool the next time it is requested.

Identifying Programs Associated with a Subschema

If your site updates the dictionary every time a program is compiled, the dictionary will contain the necessary information to identify the programs associated with a modified subschema.

If this information is stored in the dictionary, you can run IDMSRPTS Program Cross-Reference Listing report. For each program associated with a subschema, the report lists:

- Name
- Version number
- Date last compiled
- Number of times compiled
- Language

Note: For more information about IDMSRPTS, see the *CA IDMS Utilities Guide*.

If your site does not update the dictionary when a program is compiled, such information must be maintained manually.

Modifying or Deleting a Subschema

Modifying a Subschema

When You Might Want to Make This Change

There are several modifications you may want to make to the subschema definition itself (other than modifications to set, area, and record definitions). These modifications include:

- Description
- Program registration
- Authorization
- Usage
- Information on transferring statistics
- Logical record currency
- Security
- User-defined information (class/attribute and user-defined comments)

What Components are Affected

The definition of the subschema as it resides in the dictionary is affected by such modifications.

Example

In the following example, program registration has been turned on. This requires that all programs using this subschema be registered with the named subschema in order to be compiled against it.

```
modify subschema empss01
  program registration required is on.
generate.
```

Deleting a Subschema

When You Might Want to Make This Change

If there is no longer a need for a subschema, you may want to delete it.

What Components are Affected

The subschema source is affected by the deletion of a subschema. The subschema load module is not affected.

Considerations

- When you delete a subschema, programs associated with that subschema can no longer be compiled. You must associate each program with a new subschema.
- If you have not specified DELETE IS ON in the SET OPTIONS statement, the subschema load module is not automatically deleted when you delete the subschema definition. You must explicitly delete the associated load module.

Example

In the following example, the subschema EMPSS01 is deleted.

```
delete subschema empss01.
```

Adding, Modifying, or Deleting a Record

What Components are Affected

The record definition portion of the subschema is affected by such a modification.

Considerations

- If you include a new record in the subschema and that record participates in a mandatory automatic set, you have to include the owner of that set in the subschema (or the set itself) so that application programs using the subschema can store occurrences of the new record.

  When you regenerate the subschema, you will receive a notice of an access restriction.

- If you modify a record so that some elements are omitted, you may have to modify and regenerate maps for online programs as well as the programs or dialogs themselves.
- If you add a record that is stored in an area not currently participating in this subschema, you must add that area to the subschema if a program is to access the new record. You will receive an error on generation if the area is not added.

Adding, Modifying, or Deleting a Set

What Components are Affected

The set definition portion of the subschema is affected by such a modification.

Considerations

- If you do not add the set owner and member record types to the subschema, your program cannot access the new set.
- If you add the set and the set member record type but not the owner record type, the application program will not be able to obtain the owner of the set.
- If you add the set and the set owner record type but not the member record type, the application program will not be able to walk the set.
- If you delete a set but not its owner or member record type, currency will not be maintained for that set and, although an application program can access the owner, it cannot walk a set to obtain all members. In addition, the application program cannot connect a member into the set, and, if the set is mandatory automatic, the application program cannot store a new record occurrence.

Adding, Modifying, or Deleting an Area

When You Might Want to Make This Change

You can add areas to or delete areas from a subschema or make a modification to an existing area. Normally this is done in conjunction with adding or deleting a record or index structure stored in that area or to move records into a new area for performance reasons.

What Components are Affected

The area definition portion of the subschema is affected by such a modification.

Considerations

- If you modify the usage mode so that a mode is no longer allowed, you may have to modify the READY mode of your program to match.
- If you modify the default usage mode, you should check programs using the subschema to see that there is no conflict.
- If you delete an area, make sure that there are no records or indexes mapping to that area still in the subschema.
- If an area is renamed or deleted, all ADS dialogs that use the subschema must be recompiled if they use neither READY ALL nor DBMS autoready.

Adding, Modifying, or Deleting a Logical Record or Path Group

What Components are Affected

Only the definition of either the logical record or a path group is affected.

Considerations

- If you modify a logical record so that some elements are omitted, you may have to modify and regenerate maps for online programs as well as the programs themselves.
- If you remove a path group from the subschema, you must modify and recompile any program or dialog associated with that subschema using the deleted path group.
- If you have changed the selection criteria in a path, you need to modify the program requests in programs or dialogs associated with that subschema.

Note: For more information about Logical Record Facility, see the *CA IDMS Logical Record Facility Guide*.

Chapter 35: Space Management

This section contains the following topics:

Space Management

Definitions of Areas and Pages

A CA IDMS database contains one or more areas. Each *database area* is a named subdivision of addressable storage in the database. A CA IDMS area is subdivided into *database pages*. Most database pages are used to hold actual record occurrences (or rows). Some pages are reserved by CA IDMS for space management.

Note: Record occurrences and rows of an SQL-defined table are stored in the same way in a CA IDMS database. For simplification, the term record occurrence will be used to indicate both row and record occurrence, and record type to indicate both table and record type.

Definition of Database Key

Each record occurrence in a CA IDMS database is uniquely identified by a *database key (db-key)* that specifies the physical location of the occurrence. Database keys are used as pointers to related record occurrences or index records.

The format of a database key can vary from database to database. The variable format of the db-key allows you to tailor space management factors to different processing requirements.

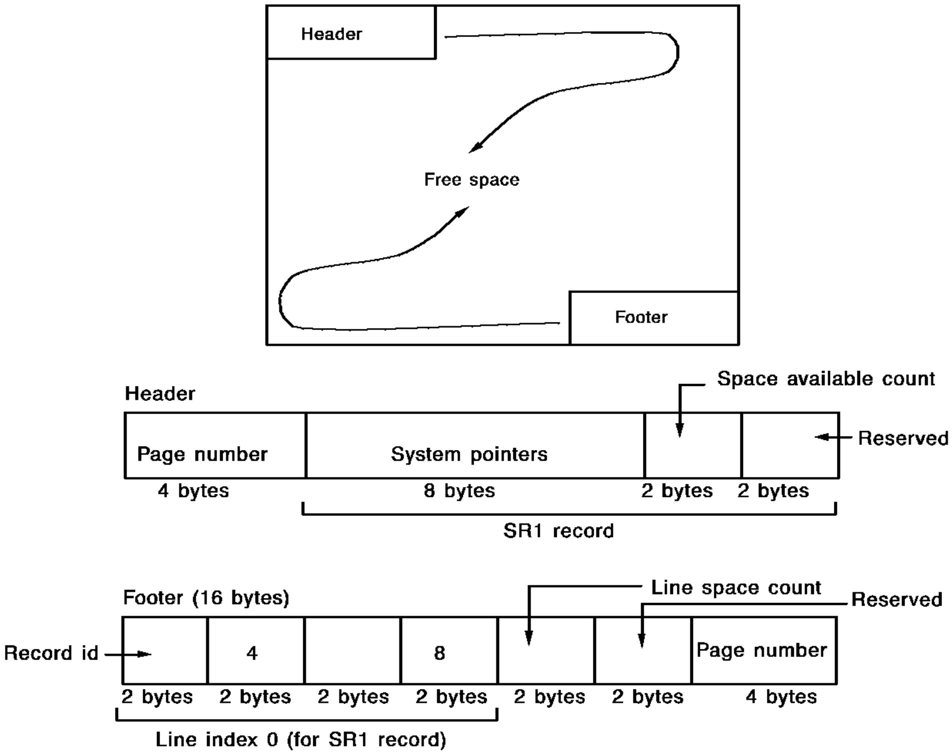
Database Pages

Size of Database

A database can have from 2 to 1,073,741,822 pages. Each area contains pages of equal size. Each page can contain up to 32,756 bytes of data. For details, see 35.3, “Database Keys". Database pages are mapped to BDAM, or DAM blocks, or VSAM control intervals (for details, see Chapter 17, “Allocating and Formatting Files"). Each database page is identified by a unique *page number* and data transfers are accomplished one page at a time.

Page Format

All database pages, regardless of size, have a header and footer with the same general format as shown in the following diagram. A database page always has a header at the beginning of the page and a footer at the end; free space is in the middle.



Header

The header occupies the first 16 bytes of each page and is formatted as follows:

- **Page number** (4 bytes)—A unique, system-assigned number of the page.
- **SR1 system record** (12 bytes)—An SR1 record is stored on each page during initialization by the FORMAT utility. Each SR1 record contains the **space available count** (that is, the number of bytes of free space on the page).

Footer

The footer occupies the last 16 bytes of each page and is formatted as follows:

- **Line index 0** (8 bytes)—Identifies the location and length of the SR1 system record
- **Line space count** (2 bytes)—Number of bytes used for line indexes and the footer
- **Filler** (2 bytes)—Reserved space
- **Page number** (4 bytes)—The unique system-assigned number of the page

Note: Numeric fields maintained by CA IDMS are in binary format, although this manual represents them as decimal numbers.

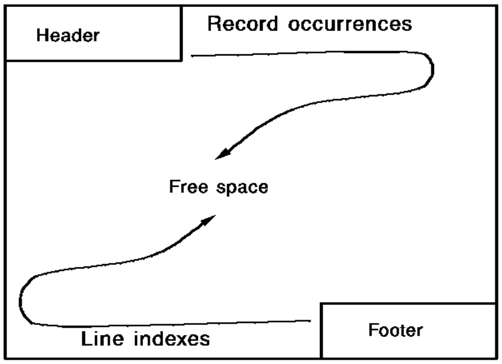
To simplify the illustrations, the page size (800 bytes) in the figures of this manual is unusually small.

Database Page Layout

Except for the header and the footer, pages are filled with the following entries:

- **Record occurrences**—The actual record occurrences are positioned on the page from top to bottom immediately following the header. Each occurrence consists of a **prefix** (containing pointers) and a **data portion**. A page can hold from 3 to 2,727 record occurrences depending on user specification (for details, see 35.3, “Data base Keys".)
- **Line indexes**—The line indexes identify the locations of record occurrences on the page and are positioned on the page from bottom to top, immediately preceding the footer. A page contains one line index per record occurrence on the page. Each line index has the following format:
  - **Record id** (2 bytes)—Identification of the record type
  - **Displacement** (2 bytes)—Location of the record occurrence relative to the beginning of the page, where the first byte on the page is position 0
  - **Record length** (2 bytes)—Length of the entire record occurrence stored on this page (data plus prefix) in bytes
  - **Prefix length** (2 bytes)—Length of the prefix portion of the record in bytes

Record occurrences are added from the top down; line indexes from the bottom up. Free space is always in the middle.



Record occurrences

Prefix	Data
4 bytes per pointer	Size according to user specifications

Line index (8 bytes)

Record id	Displacement	Record length	Prefix length
2 bytes	2 bytes	2 bytes	2 bytes

Database Keys

Identify Each Record Occurrence

Each record occurrence in a CA IDMS database is uniquely identified by a **database key** (**db-key**), which indicates the occurrence's physical location in the database. A db-key is assigned when a record occurrence is stored in the database. The db-key never changes as long as the record remains in the database (that is, until the record is erased or until the database is unloaded and subsequently reloaded).

Used as Pointers

Database keys are used as pointers to related record occurrences or index records. As such, database keys are found in the system-maintained prefixes that precede the data portion of the record occurrence. For example, a record occurrence's prefix may contain the database keys of the next, prior, and owner records of the chained set in which that occurrence is a member.

A db-key is a 4-byte (32 bit) binary number. The Database Management System (DBMS) creates a db-key for a record occurrence by concatenating the following numbers:

- **Page number**—The page on which the record occurrence is stored
- **Line number**—The position of the record occurrence's line index on the page relative to the other line indexes, where the line index for the SR1 record is line index 0

Db-key Format

The db-key format is variable. The number of bits reserved in the db-key for the page number and line number, respectively, can vary from one physical database to another, as long as the total number of bits used is 32. You identify the db-key format to be used by specifying the maximum number of record occurrences to be stored on one database page in the CREATE SEGMENT statement.

Default Db-Key Format

In the default db-key format, 24 bits are allocated for the page number and eight bits for the line number. This format allows a maximum of 16,777,214 pages in the database, with each page containing up to 255 record occurrences.

Variable Format

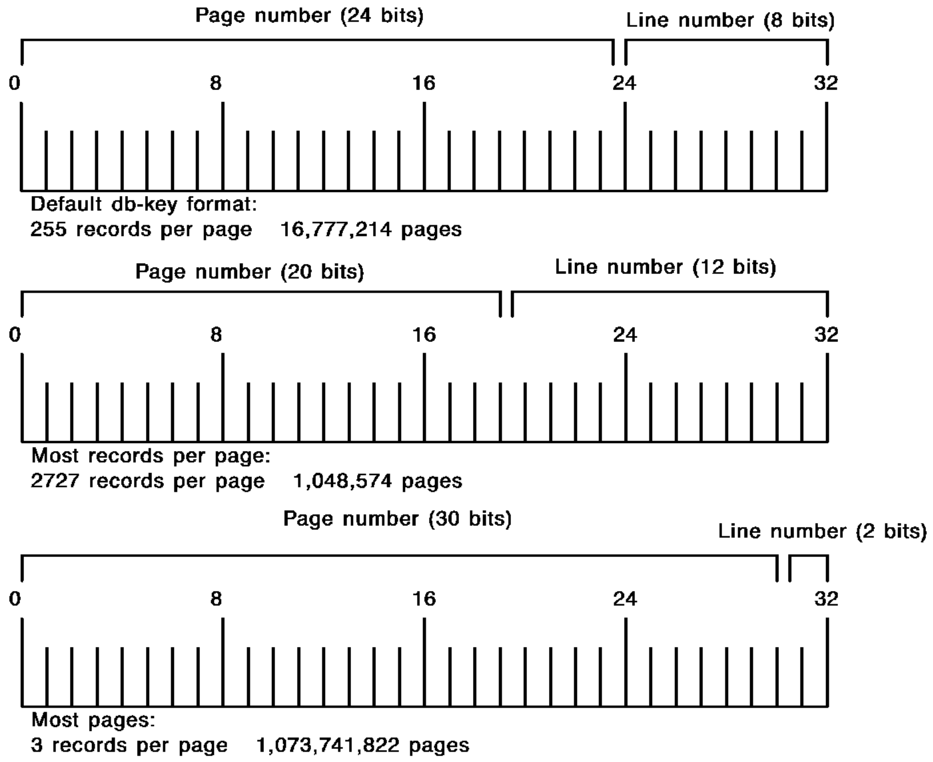
The variable format of the db-key allows you to tailor space management factors to different processing requirements. For storage of small records, specify a database with many record occurrences per page and a smaller number of pages. For storage of large records, specify a database with few record occurrences per page but a large number of pages. For these different requirements, adjust the db-key format as follows:

- *To allow more record occurrences per page,* increase the number of bits for the line index. (The line number must be from 2 to 12 bits in length.)
- *To allow more pages per database*, increase the number of bits for the page number.

As the number of record occurrences allowed on a page increases, the number of pages allowed in the database decreases. Conversely, the more pages in the database, the fewer occurrences each page can hold.

Note: The MIXED PAGE GROUP BINDS ALLOWED option for a DBNAME may be used to increase the number of records accessible in a database from a single database transaction.

The following diagram shows the db-key formats for n CA IDMS database with three possible formats: 255 record occurrences per page (the default size); the greatest possible number of occurrences per page; and the greatest possible number of pages.



Determining the Db-Key Format

Using the decimal value that you specify in the MAXIMUM RECORDS PER PAGE clause on the CREATE SEGMENT statement, CA IDMS/DB determines the db-key format, as follows:

- **To determine the total possible number of line indexes for a page**, CA IDMS/DB adds 1 to the maximum number of record occurrences per page. (This number represents line index 0, reserved for the SR1 record.)
- **To determine the size of the line number portion of the db-key**, CA IDMS/DB identifies the number of bits required to store the largest possible line index.
- **To determine the size of the page number portion of the db-key**, CA IDMS/DB subtracts the number of bits for the line number from 32 (the total number of bits in a db-key).

For example, the default number of record occurrences per page is 255. In this case, the total number of line indexes is 256 (that is, line index 0 through 255). Since the decimal number 255 takes eight bits of storage in binary format, the line number portion of the db-key for this database is eight bits, and the page number portion is 24 bits.

Note: CA IDMS uses all 32 bits of the db-key for the page number and the line number. If you want to reserve a bit in the db-key as a sign bit (that is, if you will write routines that perform arithmetic operations using the db-key sign bit), make sure that the db-keys created for your occurrences can be stored in only 31 bits.

Conversion Algorithms

Use the following algorithms to convert a db-key into individual page and line numbers:

dbkey-page **=** *dbkey***/2********bits-for-line*

dbkey-line **=** *dbkey* **- (***dbkey-page* *** (2********bits-for-line***))**

where:

- *dbkey* = the 4-byte binary database key
- *dbkey-page* = the binary database page number
- *dbkey-line* = the binary database line number
- *bits-for-line* = the number of bits for the line number in the database key

Area Space Management

What is an Area?

A CA IDMS database is divided into one or more areas. Each **database area** is a named subdivision of addressable database storage. Each area can contain one or more record types, according to varying processing requirements, but all occurrences of a particular record type must be in the same area.

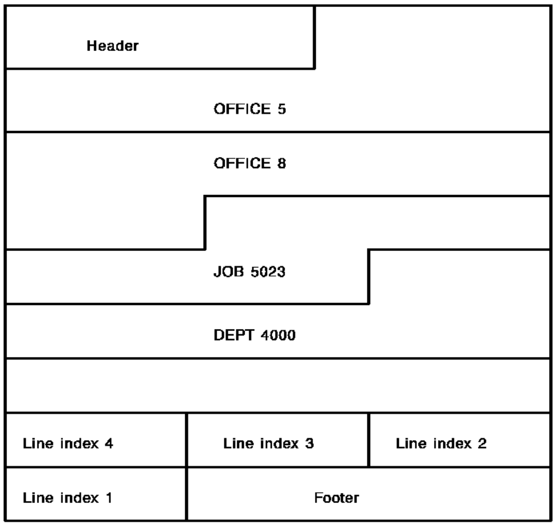
Managing Space in an Area

To manage space in an area, CA IDMS/DB keeps track of available space on each page. CA IDMS reserves selected pages called **space management pages** (**SMP**s) for this purpose. The first page in each area is an SMP. Depending on the number and size of pages in the area, CA IDMS may reserve additional SMPs throughout the area.

Since you frequently assign several record types to an area, data pages in these areas are typically filled with record occurrences of different record types and the occurrences' corresponding line indexes. For example, in the sample employee database, the DEPARTMENT, JOB, OFFICE, and SKILL records are all assigned to the ORG-DEMO-REGION area. Thus, occurrences of all of these record types can be stored on the same page.

Sample Page

The following drawing shows a sample page in the ORG-DEMO-REGION. Typically, except for the header and footer, a page in an area is filled with occurrences of different record types. Page 7130 in the ORG-DEMO-REGION area contains occurrences of the OFFICE, JOB, and DEPARTMENT record types.



Space Available

To manage space, CA IDMS/DB keeps track of the available space on each page. The space available is maintained in the following locations:

- **SR1 records**—System records in each page's header which contain the space available count for the page
- **Space management pages** (**SMP**s)—One or more system-reserved pages which contain entries that indicate whether each page (in a range of pages) is empty or full

SR1 records and space management pages are discussed separately next.

SR1 Records

Each database page in an area contains an SR1 record in the page header. Each occurrence of the SR1 record contains the space available count for that page. The SR1 record type is the owner of a set used by CA IDMS/DB to keep track of CALC records (for details, see "Storing CALC records" in Chapter 36, “Record Storage and Deletion”).

SR1 Record Format

The SR1 record is formatted as follows:

- **Next pointer for CALC set** (4 bytes)—Database key (next pointer) of the CALC record, targeted to that page, with the lowest CALC key
- **Prior pointer for CALC set** (4 bytes)—Database key (prior pointer) of the CALC record, targeted to that page, with the highest CALC key
- **Space available count** (2 bytes)—Number of bytes of free space remaining on the page
- **Filler** (2 bytes)—Reserved space

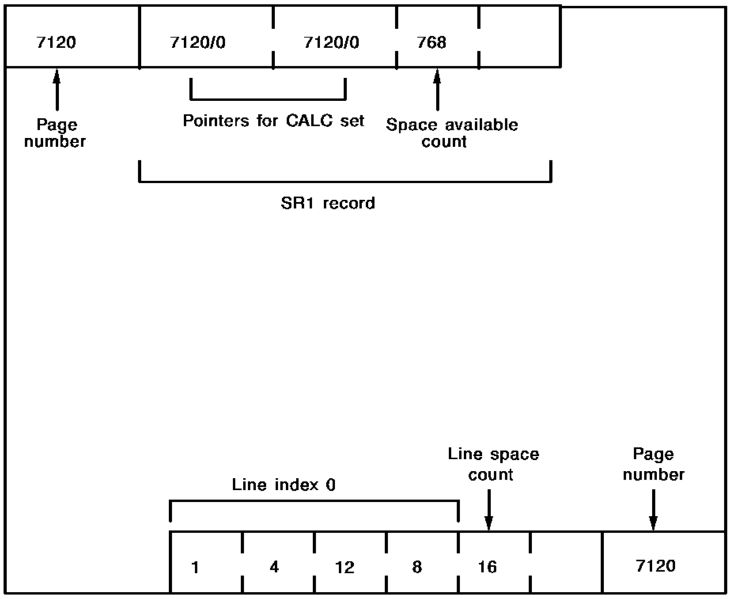
Line Index

Every **line index 0** in an area identifies the location of an SR1 record and always contains the following values:

- record identification = 1
- displacement = 4
- record length = 12
- prefix length = 8

The following diagram shows an empty page in an area. This is what a page would look like after initialization by the FORMAT utility.

Note: The *space available count* for an empty page is always the page size minus 32 (in this case, 800 - 32 = 768) and the *line space count* for an empty page is always 16. The CALC set pointers in the SR1 record on an empty page point back to the SR1 record itself since it is the only record in the set.



Space Management Pages

What is a Space Management Page?

CA IDMS reserves selected pages, called **space management pages** (**SMP**s), to keep track of the available space on each page. These pages are filled with 2-byte items called **space management entries**. Each space management entry, depending on the entry's relative position on the page, corresponds to a page in the area. The first entry corresponds to the space management page itself, the second entry to the first page following the space management page, and so on.

Number of Pages Managed by SMP

The number of pages managed by one space management page is the page size minus 32 (header and footer) divided by 2 (two bytes per space management entry).

For example, a space management page for an area whose page size is 800 bytes holds 384 entries. The first entry is for the space management page itself. If the area contained 900 pages, the area would require three space management pages. The first space management page would be the first page in the area, the second would be the 385th page, and the third would be the 769th page.

FORMAT Utility Initializes SMP Entries

For pages that will contain record occurrences, the **FORMAT utility initializes space management entries** to a value equal to the page size of the area minus the number of bytes used by the header and footer (that is, the amount of usable space on each page). The first space management entry is for the space management page and is initialized to zero. In the previous example, the space management entries for data pages would be initialized to a value of 768.

Accessing Space Management Pages

After initialization, space management pages are accessed only in the following situations:

- **STORE command**—If CA IDMS/DB cannot store a record occurrence on the target page because insufficient space exists on that page, the space management page is consulted for the next page that has sufficient space. Further, if the space available count field on the target page shows that more than 70 percent of the usable space is used, the space management page is accessed and the space management entry is changed to the actual space available. Also, if CA IDMS/DB uses the last available line index on a page to store a record, a halfword of 2 is entered in the space management entry, indicating that the page is **logically full**.
- **ERASE command**—When the actual space available for a page is shown in the space management entry (that is, when the page is more than 70 percent full) and a record occurrence is deleted from the page, CA IDMS/DB accesses the space management page and does one of the following:
  - If the page is still more than 70 percent full, CA IDMS/DB moves the new space available count from the page to the space management entry.
  - If the page is now less than 70 percent full, CA IDMS/DB reinitializes the space management entry to the value of the page size minus the length of the header and the footer (that is, the decimal value 32).

Actual Space Available

The actual space available for each page is not maintained constantly to avoid accessing the space management page each time a record is stored or erased. Instead, a page is considered empty (for space management purposes) until either of the following conditions occurs:

- A store operation for a record occurrence puts the space used over the 70 percent threshold.
- All line indexes on that page have been used (that is, the page is logically full).

A page returns to the empty status when an erase operation puts the space used back below the 70 percent threshold.

Consequently, unless a large enough page size is specified, CA IDMS/DB might attempt to store records that will not physically fit on a page.

Suppose, for example, that a page is 60 percent full and marked as empty in the space management page, and that a record occurrence being stored is 45 percent of the page size. Using information maintained in the space management page, CA IDMS/DB would determine that the record occurrence could fit on the page, when it could not.

To ensure that CA IDMS/DB can successfully store all records, *specify a page size that allows CA IDMS/DB to store the largest fixed-length record on 30 percent of the page.*

Determining Minimum Page Size

Use the following algorithm to determine minimum page size:

```
min-page-size = ((record-length + 8) / 0.30) + head-foot-length
```

where:

- *min-page-size* = the decimal value of the minimum page size
- *record-length* = the length of the largest fixed-length record type (data plus prefix)
- **8** = the length of the line index
- *head-foot-length* = the maximum length of a header and footer on a page; the decimal value 32

Reporting on Area Space Utilization

The PRINT SPACE utility statement reports on:

- Space utilization based on the contents of the SMPs
- With the FULL option, space utilization based on the actual contents of each database page (using the space available count)

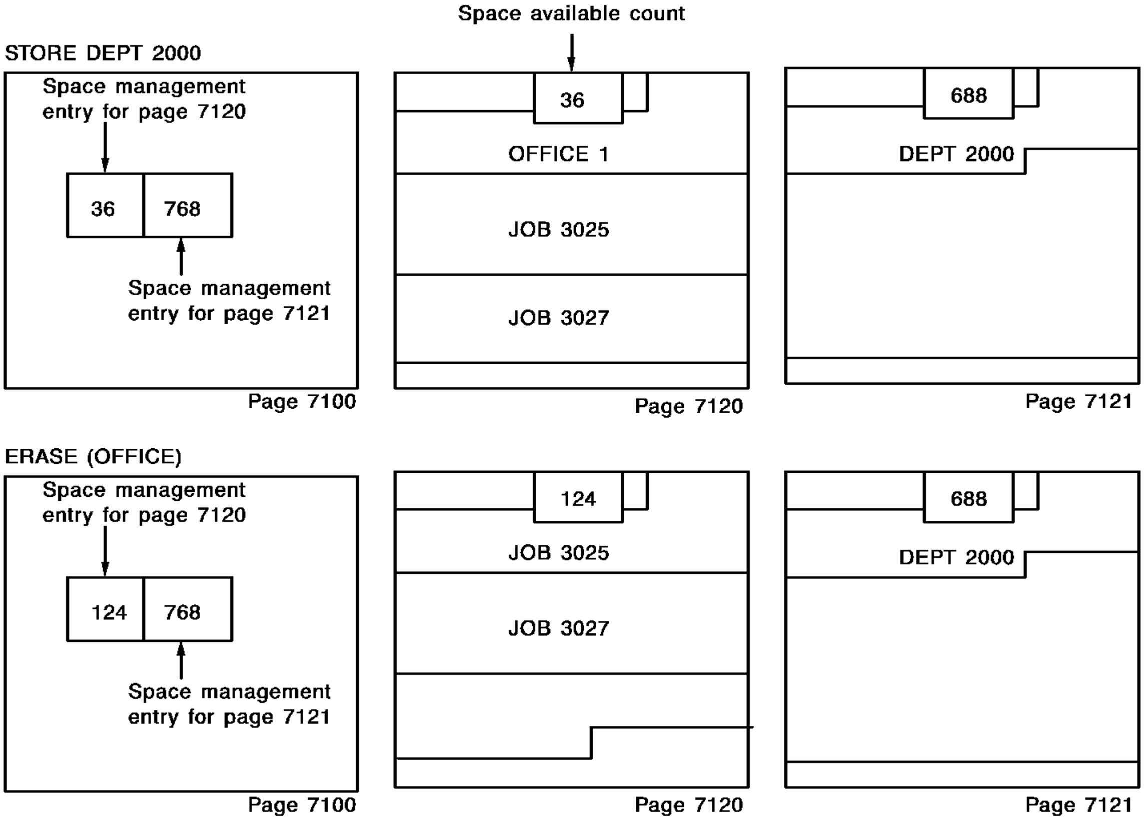
Use of the Space Management Page

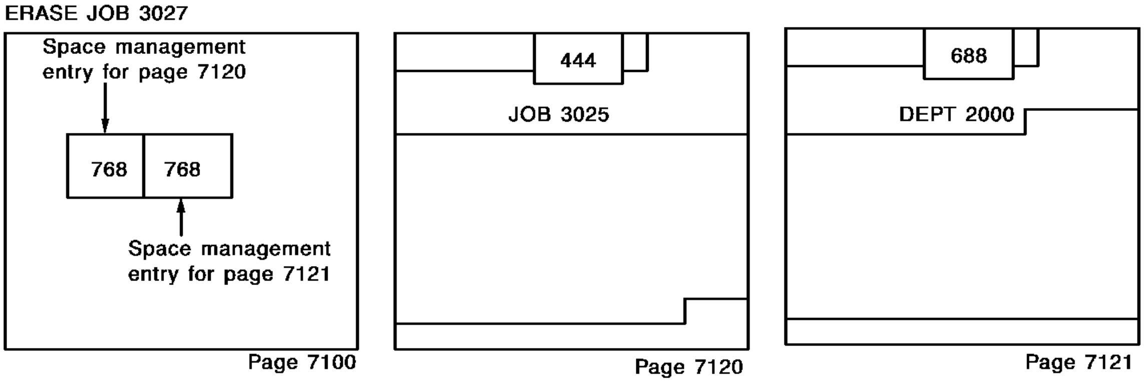
The following diagram shows the use of the space management page.

CA IDMS/DB changes the space management entry for page 7120 from 768 (the page size minus 32) to 36 (the actual number of bytes left on page 7120) after storing the JOB 3027 record. Thus, after consulting the space management page, CA IDMS/DB knows that it cannot store the DEPT 2000 record (72 bytes long) on page 7120 because of insufficient space, and stores it on the next page.

When the OFFICE 1 record is deleted from page 7120, the page is still more than 70 percent full, so CA IDMS/DB moves the value 124 (the actual amount of space available) to the space management entry.

When the JOB 3027 record is deleted, however, page 7120 is less than 70 percent full and the space management entry is reinitialized to 768 bytes.




ERASE JOB 3027
Space management entry for page 7120
768
768
Space management entry for page 7121
Page 7100
444
JOB 3025
Page 7120
688
DEPT 2000
Page 7121

Chapter 36: Record Storage and Deletion

This section contains the following topics:

Record Storage

Determining the Target Page

To store a record in the database, CA IDMS/DB first determines a *target page*. Storage mode specifications govern the selection of the target page, as follows:

- In **CALC** storage mode, CA IDMS/DB calculates the number of the target page by executing a randomizing routine against the CALC key.
- In **VIA** or **CLUSTERED** storage mode, which is used to store related record occurrences (or rows) on the same page or on as few pages as possible, CA IDMS/DB determines the number of the target page from:
  - For non-SQL, the number of the page that contains the current record of the VIA set
  - For SQL, the referenced row of a clustered constraint
- In **DIRECT** storage mode, the user explicitly specifies the target page. (Note that if you specify the value -1, the target page is the first page assigned to the record type.)

Storing the Record Occurrence

If the target page has sufficient space to store the entire record occurrence (fixed-length uncompressed records) or the record's minimum root, CA IDMS/DB then stores the record occurrence on the target page. If the target page does *not* have sufficient free space to store the record occurrence, CA IDMS/DB stores the record occurrence on the next page that has sufficient space. The search for free space always proceeds in a forward (higher database key) direction. If the end of the area (or the page range assigned to the record type) is reached before space is located, the search wraps around to the beginning of the area (or the page range assigned to the record type).

After identifying the first available free page, CA IDMS/DB performs the following operations to store a record occurrence:

- **Creates a line index** and positions it at the end of the free space or an unused line index.
- **Positions the prefix and data** (as retrieved from the program variable storage) at the beginning of the free space.

  When storing a fixed-length uncompressed record, CA IDMS/DB places the entire record occurrence on the target page. When storing a variable-length record occurrence, CA IDMS/DB places as much of the record occurrence as possible on the target page. (For details, see 36.1.3, "Storing Variable-Length Records".)
- **Updates the space available count** in the header and the **line space count** in the footer.
- **Updates the record's pointers** as follows:
  - Updates the pointers for all user sets in which the record is an automatic member
  - Sets the pointers to null (-1) for all sets in which the record is a manual member
  - Sets the pointers to the database key of the object record itself for all owner records (indicating an empty set)
  - For SQL, sets the pointers to null (-1) for linked constraints in which the table is the referencing table if one or more columns of the foreign key are null; otherwise, sets the pointers to the db-keys of related rows
  - For SQL, sets the pointers to the database key of the object row itself for linked constraints in which the table's the referenced table
- **Updates the record's CALC set pointers** (if any).
- **Updates the pointers in all other records affected** by the stored record's automatic (and CALC, if applicable) set connections.

  For example, if record B2 is being stored between records B1 and B3 in set A-B, B2's next pointer is set to B3's database key, while B2's prior pointer is set to B1's database key. Additionally, B1's next pointer is changed from B3's database key to B2's, and B3's prior pointer is changed from B1's database key to B2's.

Storing CALC Records

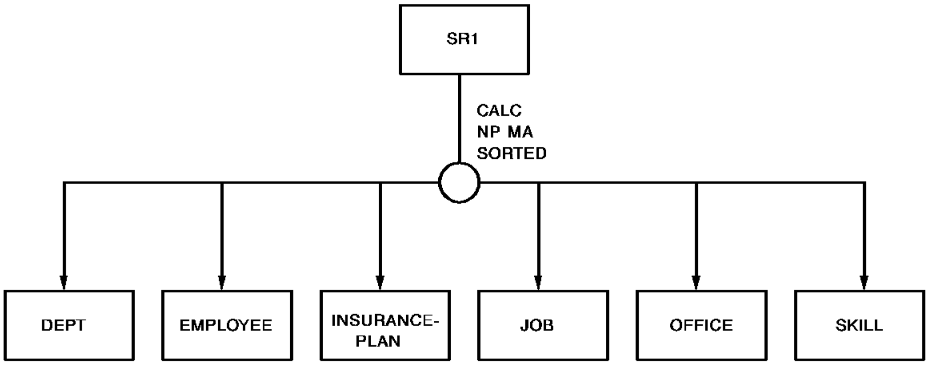
Stored On or Near Calculated Page

CA IDMS/DB stores records that have a storage mode of CALC on or near the page calculated from the record's CALC key (a schema-specified symbolic key). CA IDMS/DB uses the system-owned **CALC set** to keep track of all CALC records that randomize to a specific page. The CALC set's owner is the system-owned SR1 record type. The CALC set's members are all of the user records with a storage mode of CALC. The set is sorted in ascending sequence on the CALC key of each member record occurrence.

Example of a System-Owned CALC Set

The following diagram shows the system-owned CALC set for the sample employee database.

Note: The system-owned CALC set is an internal set. It should not be included in the user's schema or in structural diagrams.



One System-Owned CALC Set Per Database

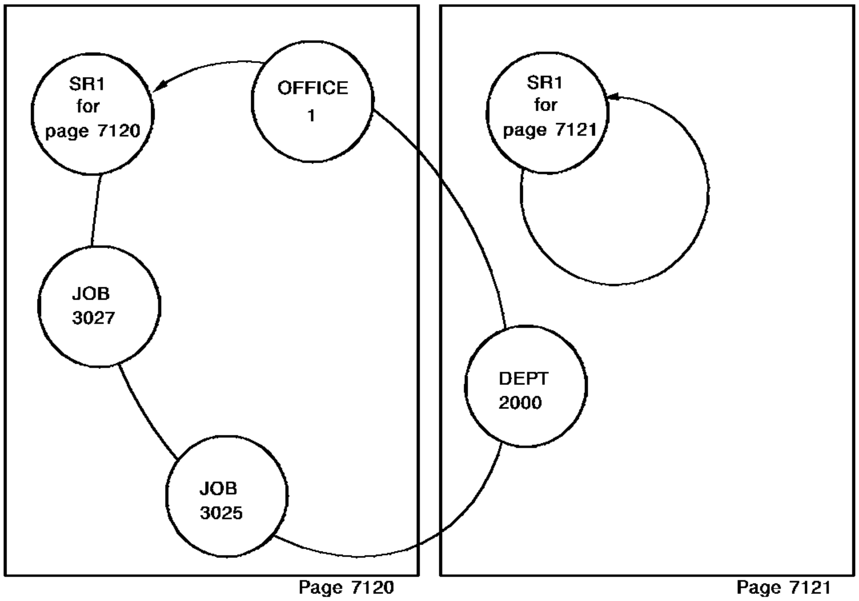
There is one system-owned CALC set type per database; there is one CALC set occurrence for each page in the area. The CALC set is sorted in ascending sequence based on the CALC key of each member occurrence.

SR1 System Record

On a page that contains record occurrences, the SR1 record on a data page owns all CALC records that randomize to that page at store time, including records that end up on another page due to overflow conditions.

The following diagram shows the format and occurrences of the CALC set on page 7120 of the sample database. The CALC set for page 7120 includes all CALC records randomized to that page.

Note: DEPT 2000 belongs to the CALC set for page 7120 even though DEPT 2000 was actually stored on page 7121 (due to lack of space on its target page).



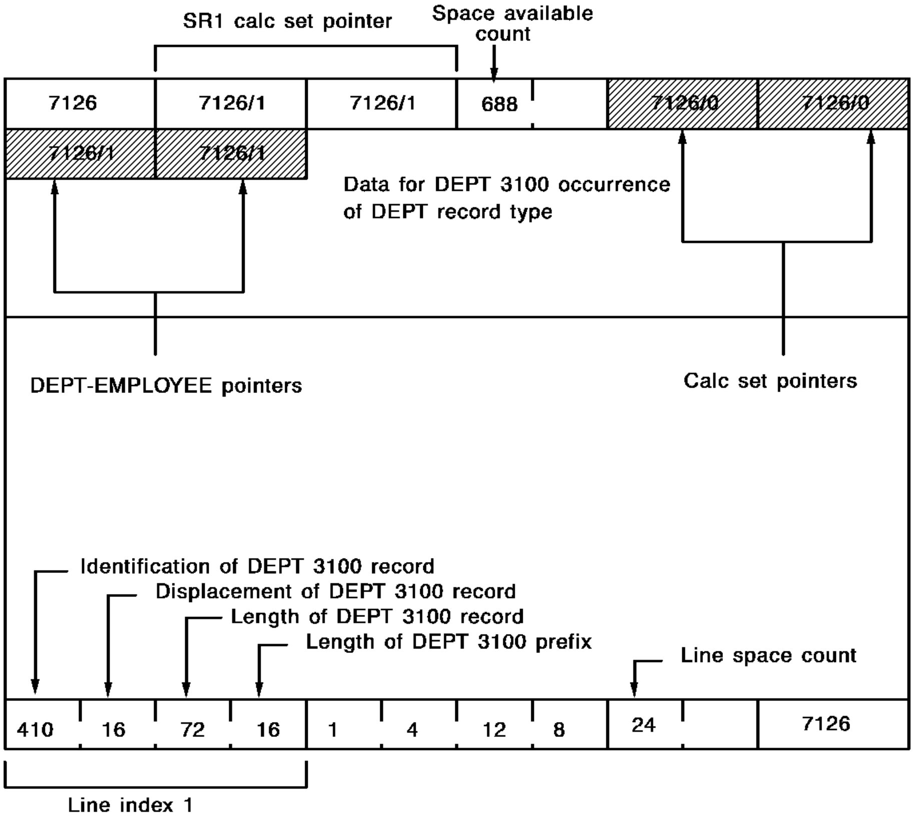
Retrieving a CALC Record

To retrieve a record occurrence stored CALC, CA IDMS/DB accepts from the user the value of the record's CALC key and calculates a page number from the key. CA IDMS/DB then enters this database page on the SR1 record and follows the page's CALC chain until either the requested record is located or a record of the same type with a higher key value is located; in the latter case, CA IDMS/DB returns an error status of 0326 (record not found) to the user.

Storing a CALC Record

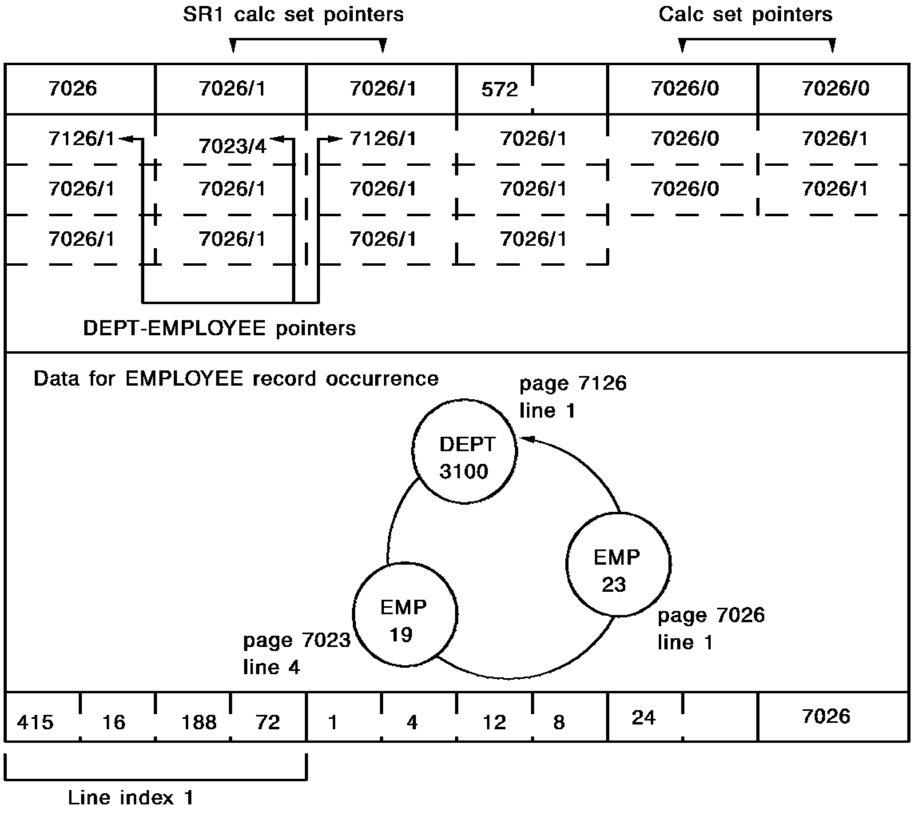
In adding the DEPT 3100 record to page 7126, CA IDMS/DB creates a record prefix (shaded portion) that consists of pointers for the CALC set and for the DEPT-EMPLOYEE set. The prefix and data (as found in program variable storage) are positioned at the beginning of the free space. A line index is created at the end of the free space. The space available count is decremented, and the line space count is incremented.

Note: The CALC pointers in the SR1 record are updated to point to the DEPT 3100 record, while the CALC pointers in the DEPT 3100 record are set to point to the SR1 record. All other pointers in the DEPT 3100 record point back to the record itself because its DEPT-EMPLOYEE set occurrence is empty.



Storing Another CALC Record

The EMPLOYEE 23 record randomizes to and is stored on page 7026. The prefix of the EMPLOYEE 23 record supplies the following information: EMPLOYEE 23 (the only member of the CALC set on page 7008) and EMPLOYEE 19 are the only members of the DEPT-EMPLOYEE set for OFFICE 3100; EMPLOYEE 19 is next of set in the DEPT-EMPLOYEE set for DEPT 3100; all of the set occurrences that EMPLOYEE 23 owns are empty.



Clustering Records

In the VIA or CLUSTERED storage mode, CA IDMS/DB stores related records together on the same page or on as few pages as possible. A record can be clustered through a chained set (a linked clustered constraint), an indexed set (a clustering index), or an unlinked constraint (SQL only).

Clustering records around a chained set

Storage Strategy

If a record has a storage mode of VIA a chained set (or CLUSTERED around a referential constraint), CA IDMS/DB uses the location of the current record of set (always the referenced row of referential constraints) to determine where to store the new record, as follows:

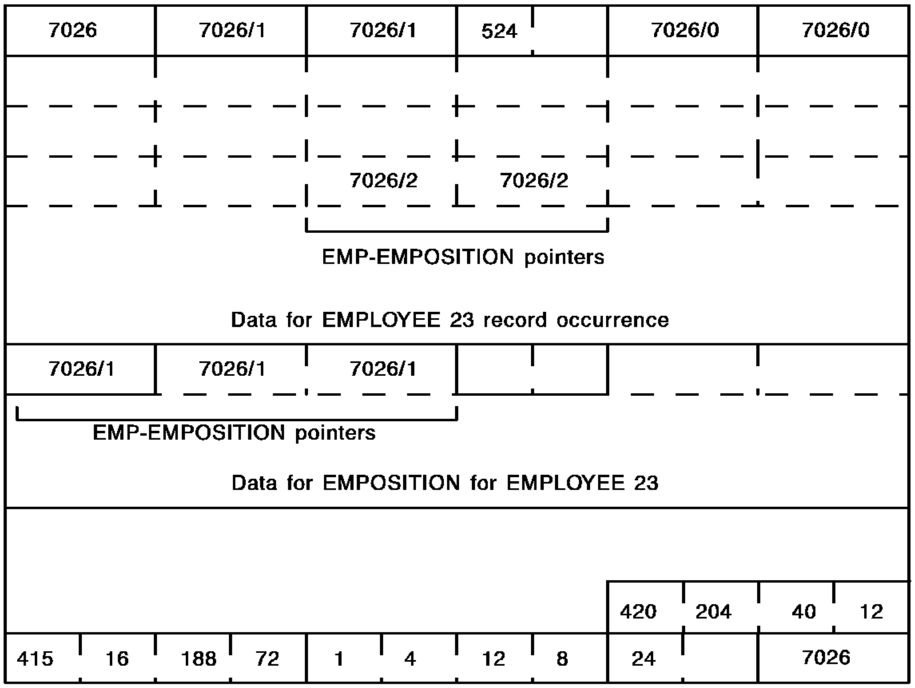
- If the current record of set is a member of the set, the DBMS stores the new record as close as possible to the current record of set.
- If the current record of set is an owner of the set, CA IDMS/DB determines where to store the member record, as follows:

If the members and owners in the specified set are assigned to the same page range, and if you have not specified displacement in the non-SQL schema...	CA IDMS/DB stores the member record occurrence as close as possible to the owner
If the members and owners in the specified set are assigned to the same page range, and you have specified displacement in the non-SQL schema...	CA IDMS/DB stores the member record occurrence as close as possible to the owner, allowing for displacement
If the members and owners in the specified set are assigned to different page ranges...	CA IDMS/DB stores the member record occurrence as close as possible to the page (within the member record's page range) that is proportional to the location of the owner (within the owner's page range)

The following diagram shows how CA IDMS/DB stores a record through a chained set. For a discussion of how CA IDMS/DB stores a record through an indexed set, see 36.1.2.2, “Storing records via an indexed set”.

Example

In this example, EMPLOYEE 23 has randomized to page 7026. EMPLOYEE 23's EMPOSITION record is stored VIA EMPLOYEE 23 on page 7026. To locate the EMPOSITION record, CA IDMS/DB applies the randomizing routine to EMPLOYEE 23 (giving page number 7026), enters page 7026 on the SR1 record, and follows the CALC set until the EMPLOYEE 23 record is located. CA IDMS/DB then obtains the EMPOSITION record through the EMP-EMPOSITION chain.



Storing records via an indexed set

Storage Order

Indexed sets can be used to store member records in a physical order that reflects the order of the member's db-key or symbolic key in the index, by defining the member record's storage mode as via (or clustered) an indexed set that is sorted on db-key or symbolic key.

Determining the Target Page

CA IDMS/DB determines the target page on which to store a member occurrence via an indexed set, as follows:

If this is the first record occurrence stored via a user-owned index set or a system-owned index with the same page range as the member record...	CA IDMS/DB determines the target page as follows: ■ If the member or owner in the set are assigned to the same page range, CA IDMS/DB stores the member record occurrence as close as possible to the owner record (allowing for record displacement if specified). ■ If the member and owner in the set are assigned to different page ranges, CA IDMS/DB stores the member record as close as possible to the page (within the member's page range) that is proportional to the location of the owner (within the owner's page range).
If this is the first record occurrence stored via a system-owned index with a separate page range from that of the member...	The target page is the low page of the member's page range
If other record occurrences have already been stored (that is, if the index is *not* empty)...	CA IDMS/DB determines the target page, as follows: ■ If the set is sorted by db-key, CA IDMS/DB finds the highest db-key of a record that is already a member of the indexed set, and uses the page specified in this db-key as the target page. ■ If the set is sorted by symbolic key, CA IDMS/DB determines the target page for the new record as follows: ■ Identifies the SR8 record that will hold the symbolic key for the new record ■ Finds the db-key of the record with the preceding or following symbolic key in the index and uses the page specified in this db-key as the target page

Example

For example, the EMP-EXPERTISE set in the sample order entry database is an indexed set, and EXPERTISE records are stored in physical-sequential order based on the value of the SKILL-LEVEL field. The non-SQL schema DDL statements necessary to specify physical-sequential placement of the EXPERTISE record are as follows:

```
RECORD NAME EXPERTISE
   LOCATION MODE VIA EMP-EXPERTISE SET ...

SET NAME EMP-EXPERTISE
   ORDER SORTED
   MODE INDEX ...
   OWNER EMPLOYEE
   MEMBER EXPERTISE ...
      DESCENDING KEY SKILL-LEVEL ...
```

In this case, CA IDMS/DB stores each EXPERTISE record as close as possible to the record with the next lower SKILL-LEVEL.

Storing Variable-Length Records

Types of Variable-Length Records

Internally, CA IDMS/DB treats the following types of records as variable-length:

	Description
Fixed-length compressed records	Records with a fixed length that are compressed through a specified compression routine. Although the length of these record types is fixed from the point of view of user programs, compression makes them internally variable.
Variable-length records	Records (either compressed or uncompressed) the length of which depends on a variable field (that is, records that contain an OCCURS DEPENDING ON clause).

Since you cannot anticipate the total length of either of these types of records, specify, in the schema, the following information:

- **The record's minimum root**—The smallest amount of the data to be stored on the record's home page
- **The record's minimum fragment**—The smallest amount of data to be stored on any additional page

Steps to Store a Variable-Length Record

Using the values specified for minimum root and minimum fragment, CA IDMS/DB performs the following steps to store a variable-length record:

1. CA IDMS/DB stores either the entire record or as much of the record as possible on the target page (provided that the space available is sufficient for the minimum root specification in the schema). This page, the first page on which CA IDMS/DB stores either the entire record or a portion of the record, is referred to as the record's *home page*; the portion of the record placed on the home page is called the *root*.
2. CA IDMS/DB stores the remainder of the record on subsequent pages, by working in a forward direction and wrapping around to the beginning of the area (or the page range assigned to the record), if necessary. Each subsequent portion of the record that exists on a separate page is called a **fragment**. No fragment except the last one will be less than the schema minimum fragment specification.

Variable-Length Indicator

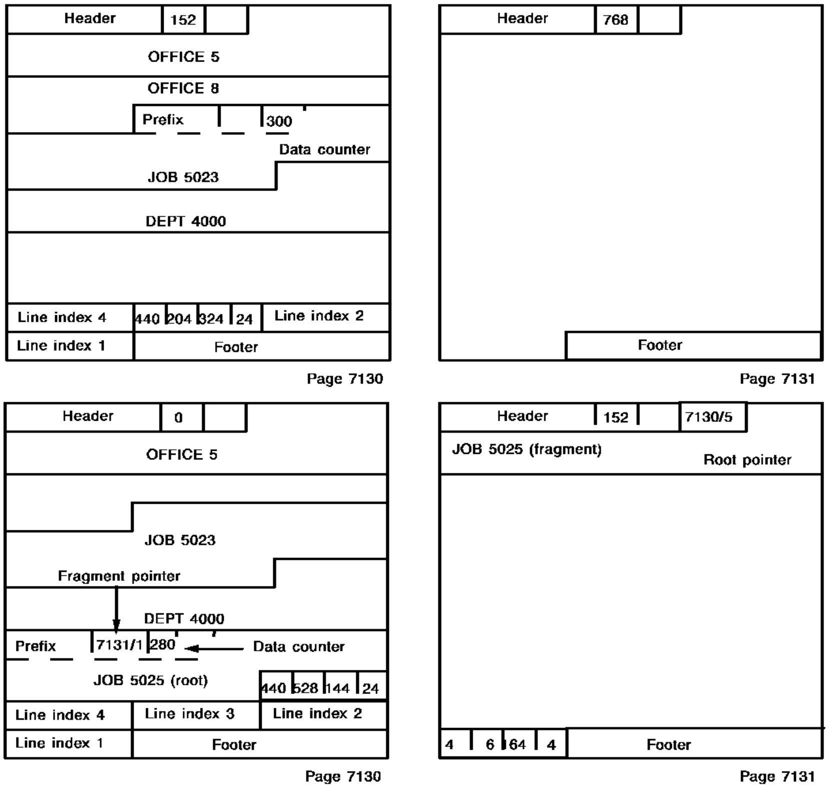
In the root, CA IDMS/DB places an extra pointer at the end of the prefix to point to the first fragment. At the beginning of the data portion of the root, CA IDMS/DB adds a 4-byte **variable-length indicator** (**VLI**) The VLI contains a 2-byte counter used to keep track of the size of the data portion of the entire record (including four-bytes for the VLI). The record-length field in the line index for a root segment contains the length of the portion of the record (prefix and data) that is stored on the home page.

SR4 System Record

Each fragment contains a one-pointer prefix that points to the next fragment; the last fragment points back to the root. Fragments are placed on a page in the same manner as any record. A fragment is considered an **SR4 system record**; the record-id field in the line index of a fragment is always set to a value of 4.

Storing a Variable-Length Record

In the following example, the JOB 5023 record fits entirely on page 7130; because the JOB record is a compressed record, it is a variable-length record and CA IDMS/DB includes a 4-byte variable-length indicator (VLI) in it, bringing the total data length of the record to 300 bytes. CA IDMS/DB cannot store the entire JOB 5025 record on page 7130; however, the page does have sufficient space for a root. CA IDMS/DB stores the root portion of JOB 5025 on page 7130 and includes a VLI, bringing the data portion of the entire record to 280 bytes. CA IDMS/DB stores the remainder of the record on page 7131 as a fragment. Note that the record-id field for the last line index on page 7131 is 4, indicating that the record is a fragment.



Returning Fragments to the Home Page

On future accesses (GET, OBTAIN, or SELECT) of a fragmented variable-length record, CA IDMS/DB may reduce the number of fragments. If the area is readied in update mode and the home page has sufficient space to hold the entire record, CA IDMS/DB returns the fragments to the page. The fragments (minus fragment pointers) are concatenated to the root and physically deleted from the pages on which the fragments were located; the fragment pointer in the root is set to point to itself. Adjustments are made to the space available count in the page header and to the record length in the record's line index.

Page Reserve

When the size of a variable-length record is increased by a DML MODIFY command, CA IDMS/DB may create additional fragments for the record. If you anticipate a general increase in the size of variable-length records in an area, specify a **page reserve** for the area to decrease the possibility that CA IDMS/DB will create fragments.

A page reserve sets aside a specified number of bytes on each database page in an area for modification of variable-length records. CA IDMS/DB cannot use this reserved space to store any kind of record.

Specify an area's page reserve in the physical database definition(s) for the area using either a CREATE AREA statement or in an area override statement within the DMCL(s) that include the area's segment. An adequate page reserve is typically 30 percent of the area's size. Use the following criteria to estimate the size of the page reserve:

- The likelihood that variable-length records will be modified
- The anticipated increase in the number of variable-length records

When you specify a page reserve, you do not affect the physical structure of the database. In fact, you can vary the page reserve for an area by using (at different times) several DMCL modules with different page reserves.

Relocated Records

Records Relocated Because of Increased Size

When increasing record sizes in areas, the RESTRUCTURE utility statement may occasionally relocate a record if the record no longer fits on its **home page**. Similarly, if a table has been altered to add one or more columns, CA IDMS/DB may relocate a row when it is next updated because it will no longer fit on its original page. The dictionary migration utility (RHDCMIG1 and RHDCMIG2) may also relocate records. When a record is stored on a new page, the relocated record is considered an **SR3 system record** and the line index created for the record on the new page contains a record id of 3.

Record Identified by SR2 System Record

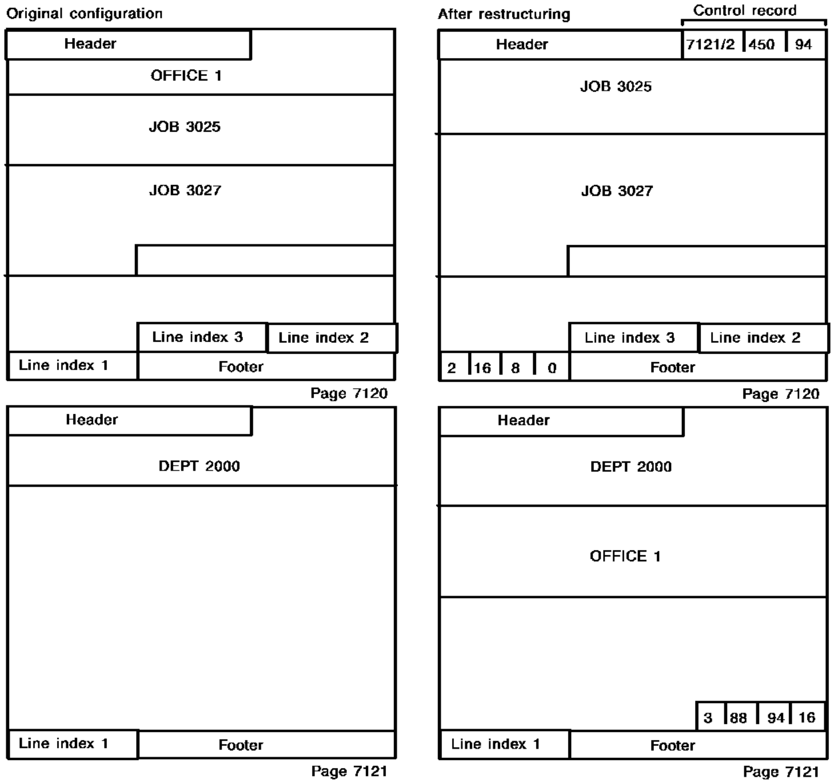
To preserve the integrity of the record's database key, CA IDMS/DB leaves an 8-byte control record (an **SR2 system record**) on the home page in place of the relocated record. The SR2 system record has a record id of 2 and contains the following information about the relocated record:

- **Database key** (4 bytes)—The pointer (db-key) to the new location of the relocated record
- **Record id** (2 bytes)—The original record id of the relocated record
- **Length** (2 bytes)—The **total length** (fixed-length records) or **root length** (variable-length records) of the relocated record

Returning Relocated Record to its Home Page

On future accesses (GET, OBTAIN, or SELECT) of a relocated record, CA IDMS/DB may return the relocated record to its home page. If the area is readied in update mode and the home page has sufficient space to hold the relocated record, CA IDMS/DB returns it to the page.

In the following example, the OFFICE 1 record, increased in size by RESTRUCTURE, is moved from page 7120 to 7121.



Record Deletion

Operations Performed

To erase a record (or delete a row), CA IDMS/DB performs the following operations:

- *Disconnects and/or erases all records owned by the object record*, depending on the nature of the ERASE DML command issued by the program (that is, ERASE, ERASE ALL, ERASE PERMANENT, or ERASE SELECTIVE).

  Note: When using SQL, the row must not be referenced by any other row.

- *Disconnects the object record from all indexed sets in which it participates as a member.*

- *Disconnects the object record from all chained sets (with prior pointers) in which it participates as a member.*
- *Deletes the record* either physically or logically, as follows:
  - If all chained sets in which the record participates as a member have prior pointers, CA IDMS/DB *physically* deletes the record.
  - If any of the chained sets in which the record participates as a member do *not* have prior pointers, CA IDMS/DB *logically* deletes the record.

    Note: If CA IDMS/DB has identified the prior record in each chained set (without prior pointers) in which the record participates (for example, walking the set), CA IDMS/DB *physically* deletes the record.

    Note: All linked clustered constraints have prior pointers.

Physical Deletion

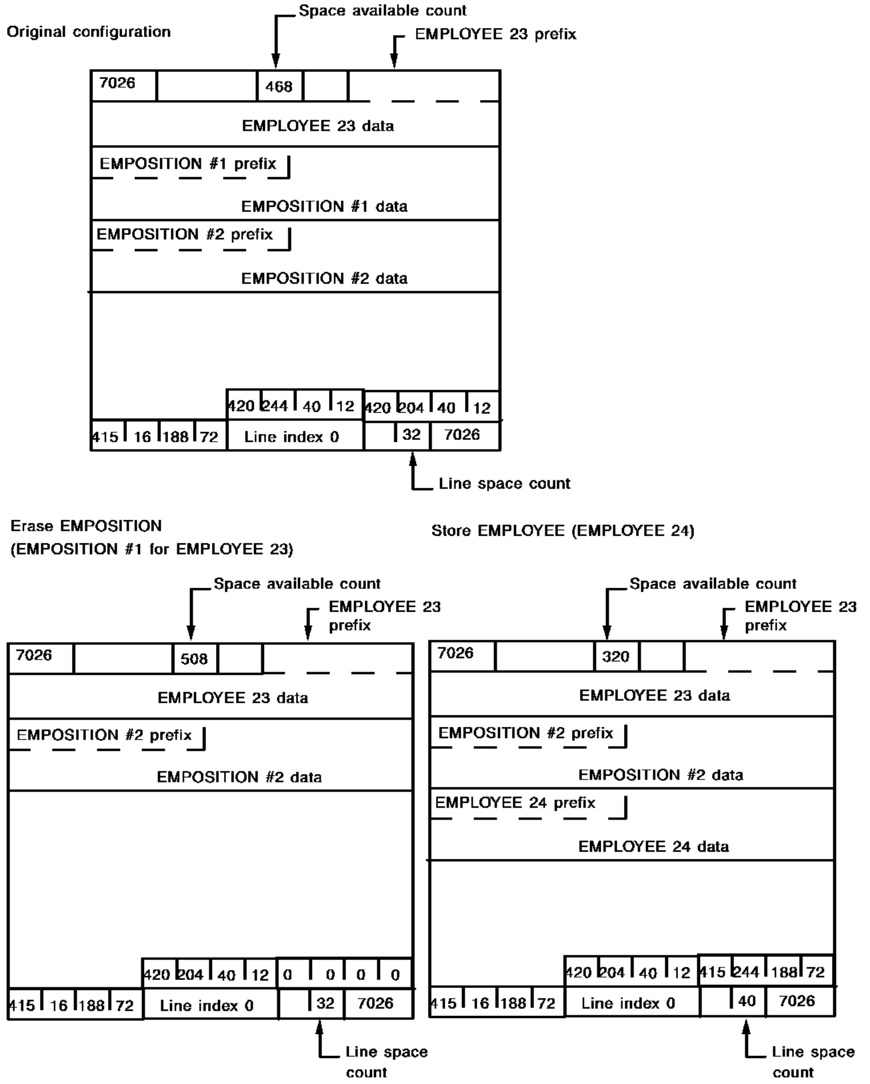
Operations Performed

CA IDMS/DB performs the following operations to physically delete a record:

1. Removes the record's data and prefix from the database.
2. Moves all records following the deleted record on the page, so that all free space remains in the middle of the page.
3. Performs the following operations, depending on the location of the record's line index on the page:
   - If the line index is contiguous with the free space on the page (that is, if the record's line index is the last index on the page), CA IDMS/DB removes the line index and updates the line space count in the footer.
   - If the record's line index is *not* contiguous with the free space on the page, CA IDMS/DB sets the record's line index to zeros.
4. Updates the space available count in the header.

Example

In this example, the first EMPOSITION record for EMPLOYEE 23 has prior pointers. In erasing the record, CA IDMS/DB removes the record completely (data and prefix), shifts the remaining EMPOSITION record up on the page, and sets the line index for the deleted record to zeros. The remaining EMPOSITION record, although now physically the second record on the page, remains as line number 3. Line index 2 is reused when a new record is added to the page.


Original configuration

Erase EMPOSITION
(EMPOSITION #1 for EMPLOYEE 23)

Store EMPLOYEE (EMPLOYEE 24)

Use of Record's Line Index

Line indexes cannot be shifted down because the position of the line index relative to other line indexes determines the line number, and changing a record's line number would invalidate the record's database key. Existing line indexes for physically deleted records are either reused as new records are added to the page (as shown in the previous diagram or removed as further deletions make them contiguous to the free space.

Logical Deletion

Pointers Deleted

To avoid consuming unnecessary time and I/O disconnecting records from sets without prior pointers, CA IDMS/DB does not physically delete the record when an ERASE command is issued. Instead, the next time CA IDMS/DB encounters a logically deleted record while walking a chained set of which the record is a member, CA IDMS/DB disconnects the record from the set, provided that the record's area was readied in update mode. Since the record prior to the logically deleted record is still current of run unit, CA IDMS/DB can update the record's next pointer and disconnect the logically deleted record. To be physically deleted, the record must have been disconnected from all sets in which the record was a member.

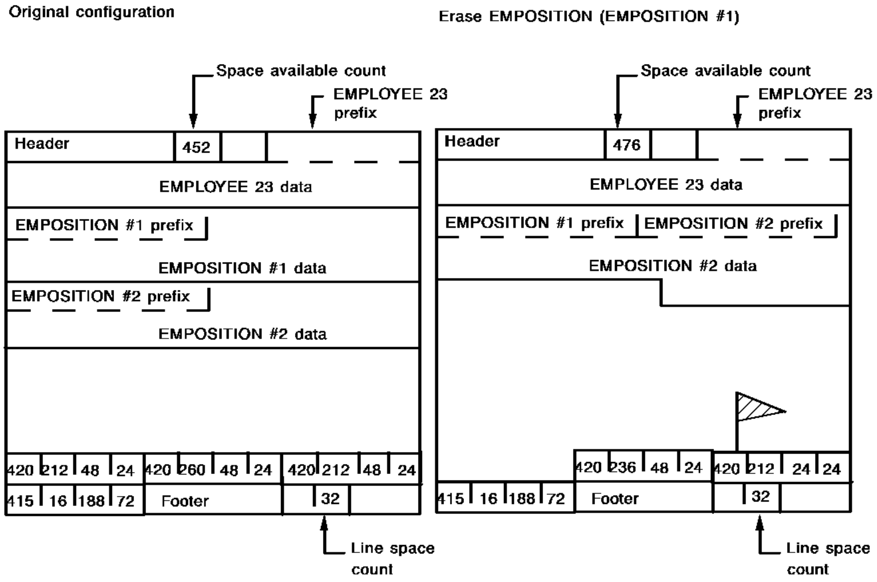
Operations Performed

CA IDMS/DB performs the following operations to logically delete a record:

- Removes the record's data from the database, but leaves the prefix
- Moves all records following the deleted record on the page, so that all free space remains in the middle of the page
- Sets the logical delete flag (the first bit) in the record id field of the record's line index
- Updates the space available count field in the header

Example

In the following example, assume that the EMPOSITION records do not have prior pointers in the EMP-EMPOSITION set. When erasing an EMPOSITION record, CA IDMS/DB removes only the data and flags the record's line index. The EMPOSITION record is logically deleted. The next time CA IDMS/DB is walking this occurrence of the EMP-EMPOSITION set in update mode and encounters the flagged record, CA IDMS/DB physically deletes the record.



Consideration

Occasionally, in recovering from an error during a store operation, CA IDMS/DB may create a logically deleted record. If CA IDMS/DB has stored a record and is in the process of making the automatic connections when CA IDMS/DB discovers an error condition (for example, no currency established in one of the automatic sets), CA IDMS/DB must erase the record being stored. If one of the chained sets to which the record has already been connected has next pointers only, CA IDMS/DB logically deletes the record.

Chapter 37: Chained Set Management

This section contains the following topics:

Overview

Physically Link Record Occurrences Together

Chained sets can be used to physically link related record occurrences together. In a chained set, a pointer in each member record occurrence's prefix contains the db-key of the logically next occurrence of the set.

Defining a Chained Set

Define a set as chained as follows:

Name	Description
Non-SQL schema definition	MODE IS CHAIN on the SET statement.
SQL schema definition	LINKED CLUSTERED on the CONSTRAINT statement. When a constraint is implemented as a chained set, the referenced table is the *owner* of the set and the referencing table is the *member*.

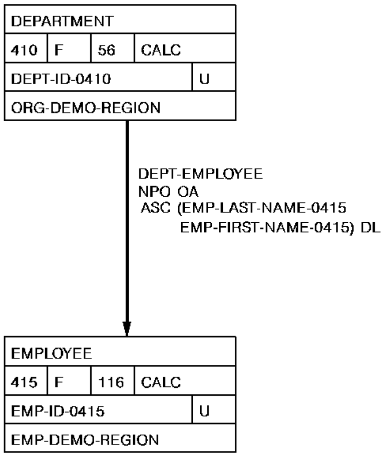
Chained Sets

Use

A chained set is used to establish a logical relationship between two or more user-defined record types and consists of an owner record type and one or more member record types.

The following diagram uses standard CA IDMS database notation to describe a chained set type; the diagram includes the name of the set, linkage options, membership options, sort sequence (if any), and sort key (if any).

This example shows a chained set (the DEPT-EMPLOYEE set) between two user-defined record types. The owner of the DEPT-EMPLOYEE set type is the user-defined DEPARTMENT record type; the member is the EMPLOYEE record type.



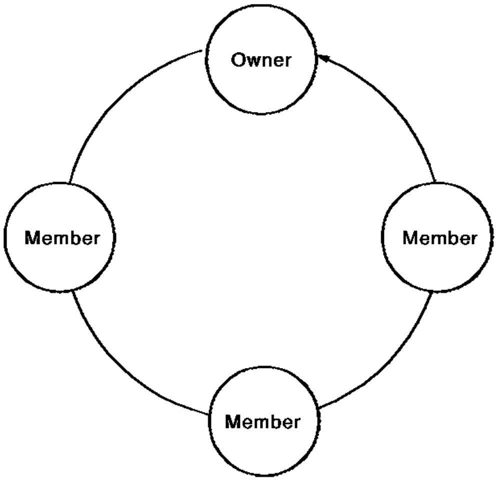
Next, Prior, and Owner Pointers

A **chained set occurrence** consists of one occurrence of the owner record type and any number of member record occurrences. The prefix of each record occurrence that participates in a set contains a **next** pointer (that is, the db-key of the next logical record occurrence in the set occurrence). Optionally, record occurrences can include **prior** pointers, which link records together in the logically prior direction, and **owner** pointers, which link member record occurrences to the owner occurrence.

Note: SQL-defined constraints implemented as a chained set always have next, prior, and owner pointers.

Basic Structure of a Chained Set Occurrence

A record occurrence in a chained set occurrence always contains in its prefix a next pointer that points to the logically next record occurrence in the set occurrence.



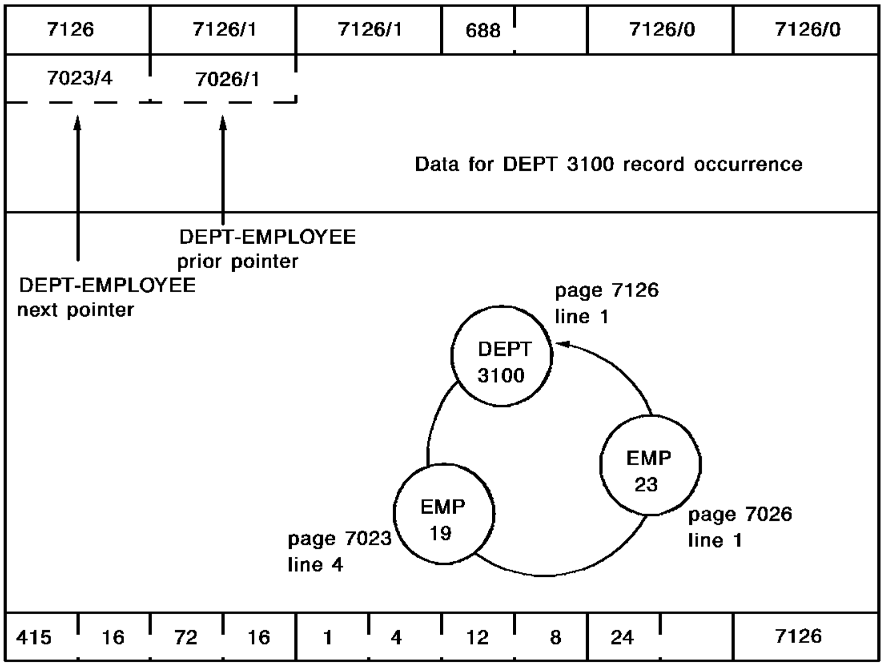
Connecting Records to Chained Sets

Operations Performed

CA IDMS/DB performs the following operations to connect a record (that has previously been stored) to a chained set:

- Updates the prefix of the record being connected to reflect the record's next, prior, and owner (as applicable) pointers in the set
- Updates pointers in all other records affected by the new set connections

In the following example, EMPLOYEE 19 and EMPLOYEE 23 have been stored on pages 7023 and 7026, respectively. Connecting each to DEPT 3100 as members of the DEPT-EMPLOYEE set affects the DEPT 3100 record on page 7126. Its prefix must be updated to point to the next and prior members of the set.



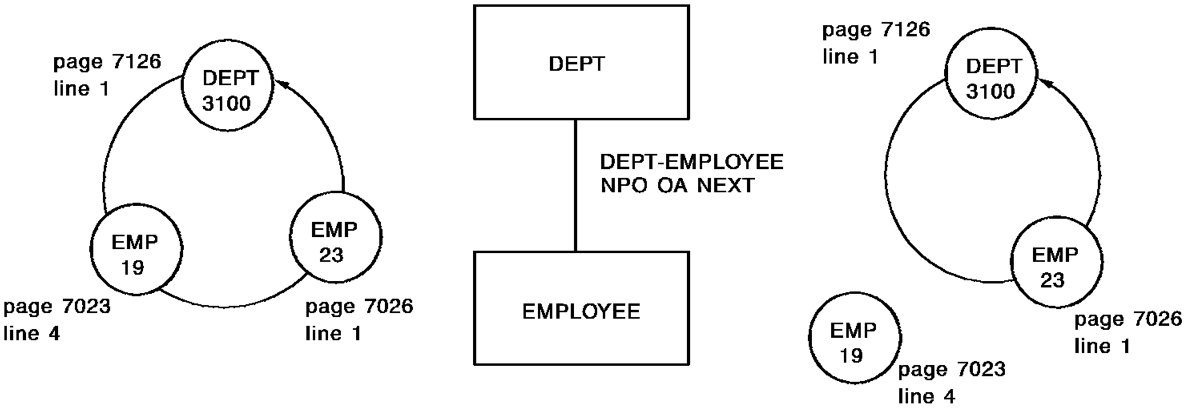
Disconnecting Records

Operations Performed

To disconnect a record occurrence from a chained set without erasing the record occurrence, CA IDMS/DB must update pointers in the current, prior, and next records, as described next:

- For the **record being disconnected**, CA IDMS/DB adjusts all the record occurrence's pointers to null (minus 1) for the set from which the record is being disconnected.
- For the **prior record in the chain**, CA IDMS/DB adjusts the next pointer for the set from which the record occurrence is being disconnected so that the prior record points to the next record.
- For the **next record in the chain** (if the set has prior pointers), CA IDMS/DB adjusts the prior pointer for the set from which the record occurrence is being disconnected so that the next record points to the prior record.

The following diagram shows disconnecting a record. The EMPLOYEE 19 record is disconnected from the DEPT-EMPLOYEE set for DEPT 3100. EMPLOYEE 19's pointers for that set are changed to null. The prior pointer in the EMPLOYEE 23 record is adjusted to point to the DEPT 3100 record, while the next pointer in the DEPT 3100 record must be adjusted to point to the EMPLOYEE 23 record.



DEPT 3100

Next	Prior
7023/4	7026/1

EMP 19

Next	Prior	Owner
7026/1	7126/1	7126/1

EMP 23

Next	Prior	Owner
7126/1	7023/4	7126/1

Before disconnect

DEPT 3100

Next	Prior
7026/1	7026/1

EMP 19

Next	Prior	Owner
-1	-1	-1

Null

EMP 23

Next	Prior	Owner
7126/1	7126/1	7126/1

After disconnect

Adjusting the Pointer

To adjust the next pointer in the prior record, CA IDMS/DB must access the prior record. In a set without prior pointers, however, CA IDMS/DB must walk the entire set to access the prior record. For this reason, prior pointers are typically included in all sets to which the DISCONNECT (or ERASE) DML command might be applied.

Retrieving Records

Walking a Set

A program using navigational DML or CA IDMS/DB in response to an SQL request can access all of the members of a chained set in the following manner: starting with the owner record occurrence, a program can use the next pointers to access each member occurrence in the chain until the program reaches the owner record again. Accessing members in a chain in this way is known as "walking a set."

Assume that the DEPT-EMPLOYEE set in the sample database is a chained set sorted by employee identification number (EMP-ID-0415). To retrieve an occurrence of the EMPLOYEE record, a program could issue the following requests:

```
MOVE '0050' TO DEPT-ID-0410
OBTAIN CALC DEPARTMENT.
OBTAIN NEXT WITHIN DEPT-EMPLOYEE SET.
```

Processing the Request

CA IDMS/DB processes this request as follows:

1. Using the value '0050' placed by the program in the DEPT-ID-041 0 field, CA IDMS/DB obtains the DEPARTMENT record with an identification number of '0050'.
2. CA IDMS/DB then finds the record occurrences pointed to by DEPARTMENT 50's next DEPT-EMPLOYEE pointer.

Retrieving an Owner

A program can issue the following request to retrieve an occurrence of the DEPARTMENT record associated with an employee:

```
MOVE '0019' TO EMP-ID-0415.
OBTAIN CALC EMPLOYEE.
OBTAIN OWNER WITHIN DEPT-EMPLOYEE SET.
```

Processing the Request

CA IDMS/DB processes this request as follows:

1. Using the value '0019' placed by the program in the EMP-ID-0415 field, CA IDMS/DB obtains the EMPLOYEE record with an identification number of '0019'.
2. If the DEPT-EMPLOYEE set has owner pointers, CA IDMS/DB uses the EMPLOYEE record's owner pointer to retrieve the owning DEPARTMENT.
3. If the DEPT-EMPLOYEE set does not have owner pointers, CA IDMS/DB uses the EMPLOYEE record's next pointer to walk the set until it retrieves the owner occurrence (that is, an occurrence of the DEPARTMENT record type).

Chapter 38: Index Management

This section contains the following topics:

Indexed Sets

Use

Indexed sets can be used to physically link related record occurrences together or to provide alternate access to a record. In an indexed set, a pointer array associated with each owner occurrence contains the db-keys of all related member record occurrences.

Types of Indexed Sets

There are two types of indexed sets:

Set	Description
User-owned	The owner of the set is a user-defined record.
System-owned	The owner of the set is a system-defined SR7 record. The location mode of an SR7 record is CALC on the set name for non-SQL defined indexes or on an internally-generated name for SQL defined indexes. There is at most one occurrence of an SR7 record for each system-owned index.

How to Define an Indexed Set

Use the following clauses on the SET statement to define an indexed set in a non-SQL schema definition:

Set	Description
User-owned	MODE IS INDEX

Set	Description
System-owned	 MODE IS INDEX OWNER IS SYSTEM

Use the following SQL statements to implement an SQL defined constraint as an indexed set:

Set	Description
User-owned	Use this clause on the CONSTRAINT statement: LINKED INDEX
System-owned	Use the CREATE INDEX statement

When you implement a constraint as an indexed set, the referenced table is the *owner* of the set and the referencing table is the *member*.

Set Order

An indexed set can have any of the following set orders: FIRST, LAST, NEXT, PRIOR, or SORTED. If it is SORTED, it can be sorted either on a user-specified symbolic key (sort key) or on the db-key of the member record occurrences.

Using SQL, the set order of a LINKED INDEX constraint is:

- SORTED, if you specify the ORDER BY clause
- Otherwise, LAST

The set order of an indexed set created using the CREATE INDEX statement is:

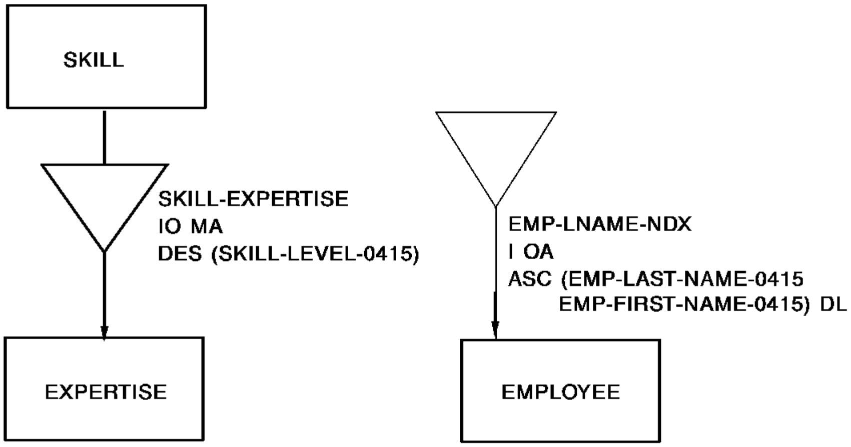
- SORTED on a symbolic key if you specify the ORDER BY clause
- Otherwise, SORTED on db-key

Notation

The following diagram uses standard CA IDMS database notation for two indexed sets, SKILL-EXPERTISE and EMP-LNAME-NDX. The descriptions of the indexed set relationships in the figure include the name of the set, linkage options, membership options, and the sort sequence and symbolic key.

The left side of the figure illustrates an indexed set (the SKILL-EXPERTISE set) between two user-defined record types. The owner of the SKILL-EXPERTISE set is the user-defined SKILL record; the member, EXPERTISE, is the indexed database record.

The right side of the figure illustrates an indexed set (the EMP-LNAME-NDX set) used to place an index on a user-defined member record type. The owner of the EMP-LNAME-NDX set is a system record, represented by a triangle; the member, EMPLOYEE, is the indexed database record.

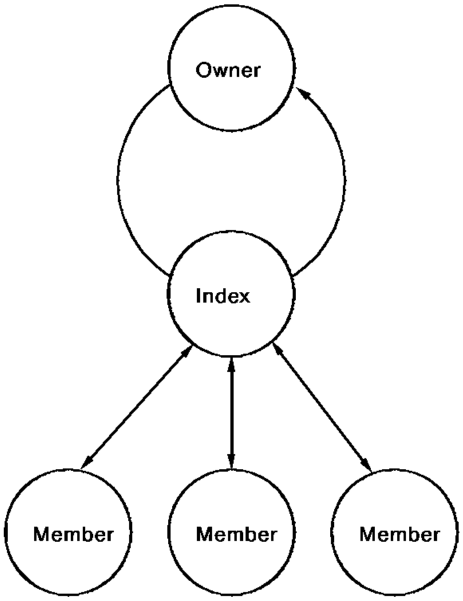


Indexed Set Occurrence

An **indexed set occurrence** consists of one occurrence of the owner record, type, an index, and any number of member record occurrences. The owner occurrence contains **next** and **prior** pointers to the index; the bottom-level of the index contains the member record occurrences' db-keys in the specified set order. If the indexed set has a user-defined owner record, each member occurrence contains an *index pointer* to the bottom-level of the index, and optionally, a pointer that links them directly to the owner occurrence. If the indexed set is system-owned, each member occurrence may *optionally* contain an index pointer.

Note: AN SQL defined linked constraint implemented as an indexed set always has owner pointers.

Basic Structure of an Indexed Set: The member record occurrences in an indexed set point to the index that is chained to the owner record by next, prior, and owner pointers. The owner record contains next and prior pointers that chain it to the index.



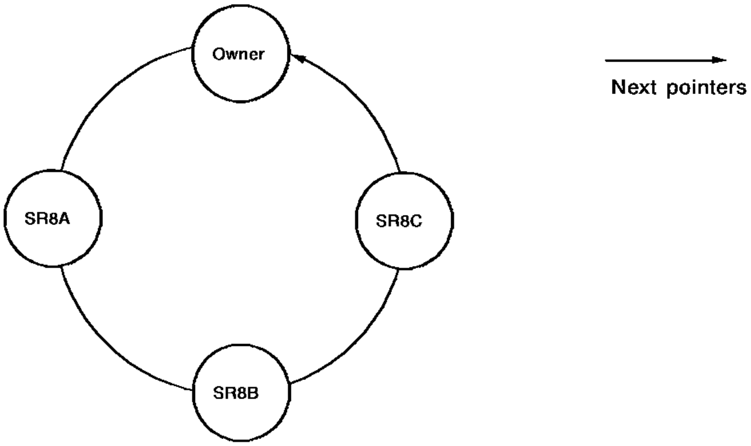
Structure of Indexes

Index Creation

The creation of an index is transparent to application programs. An index is created according to your specifications, but the actual creation and storage of the index is performed by CA IDMS/DB. An index is composed of **SR8 system record** occurrences chained (by next, prior, and owner pointers) to the owner occurrence and each other.

SR8 Records in an Index

Thus, an index is a chained set between the indexed set's owner record and the SR8 records. An index contains SR8 records chained by next, prior, and owner pointers to the indexed set's owner record. For simplicity, prior and owner pointers are not included in the next figure):

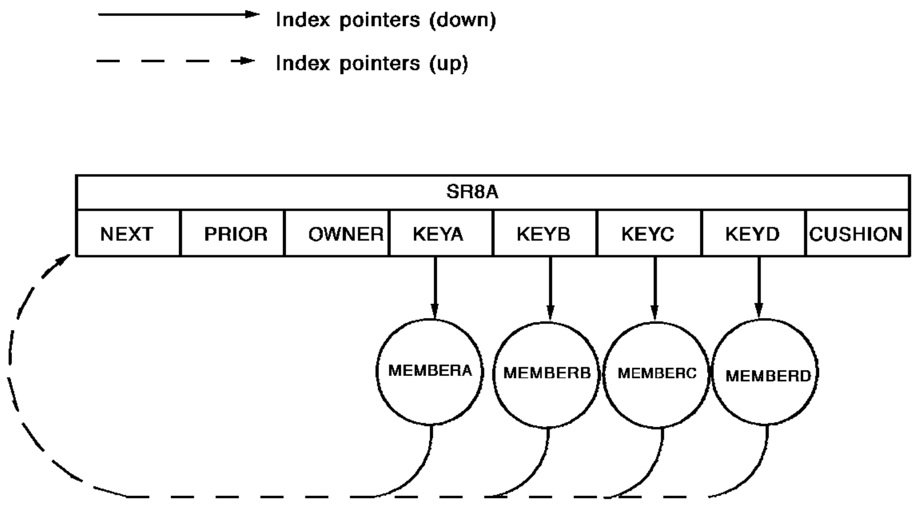


Initially, the index is composed of a single SR8 member record. When the first SR8 record is full, additional SR8 records are added to the index as chained records.

Bottom-Level SR8 Record and Database Record Occurrences

An SR8 record, shown in the following diagram, contains from 3 to 8,180 **index entries** (as specified in the schema or segment definition) and a **cushion** (that is, a field the length of the largest possible index entry).

The SR8 record in the diagram contains four entries and a cushion. Each index entry contains an index pointer that points to a database occurrence that is a member of the indexed set; each member occurrence contains an index pointer that points to that SR8 record. (Note that, for simplicity, prior and owner pointers are not included in this figure.)



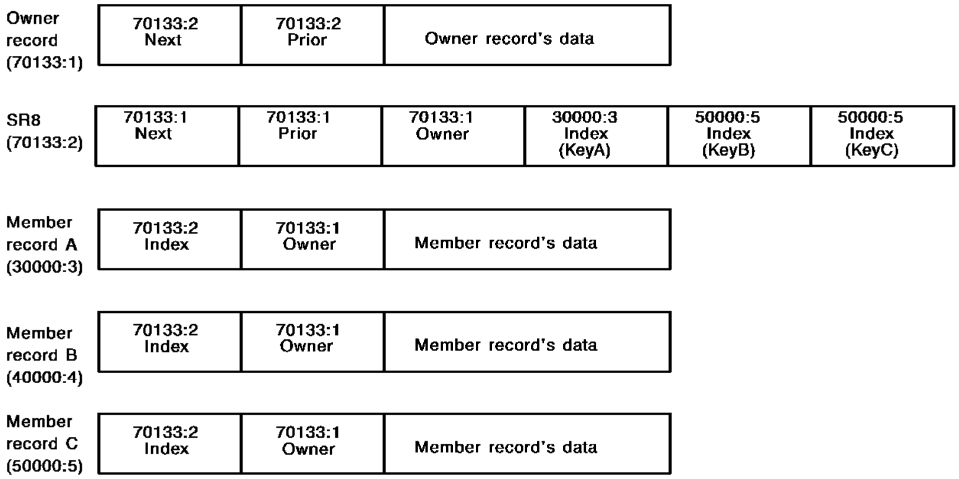
Content of an Index Entry

The actual content of an index entry depends on the indexed set's characteristics, as follows:

- **Unsorted set**—An index entry contains only the db-key of a member record occurrence.
- **Sorted set**—SR8 records for sorted indexed sets are arranged in levels to form a tree structure to facilitate a binary search. Consequently, an index entry contains the db-key of a member record occurrence or the db-key of another SR8 record occurrence. Additionally, for indexed sets sorted on a symbolic key, an index entry is composed of a db-key and a symbolic key. A symbolic key is a key constructed of one or more record elements (or columns) in the order specified in the schema (up to 256 bytes in length). (For a detailed discussion of indexed set structure for sorted indexed sets, see 38.3.2, “Connecting Members to Sorted Indexed Sets".)

Example

In this example, there is a single SR8 record chained to the indexed set's owner. The SR8 record contains three entries. Each entry contains an index pointer that points to a member database occurrence; each member occurrence contains an index pointer that points to that SR8 record. Additionally, the member occurrences contain owner pointers that point back to the set's owner.



Indexed Set with Sorted Set Order

For sorted indexed sets, you can specify that CA IDMS/DB keep the index entries within the SR8 records in ascending, descending, or mixed order according to the member record's db-key or symbolic key. You can also specify whether numeric fields should be collated so that negative values are lower than positive values (natural sequence) or whether they should collate based on their bit pattern. If you specify that an indexed set be sorted on symbolic key, you can also specify whether duplicate symbolic keys are allowed or disallowed. Even if you specify that duplicate symbolic keys are allowed, CA IDMS/DB does not store the same symbolic key more than once in the index. For example, the first time a record with a symbolic key ADAMS is added to the indexed set, CA IDMS/DB adds the symbolic key ADAMS to the index and associates the record occurrence's db-key to the key ADAMS. Later, if you add another record with the symbolic key ADAMS to the indexed set, CA IDMS/DB associates the db-key of the new record to the existing symbolic key of ADAMS in the SR8 record.

Specifying Compression

Additionally, you can specify that CA IDMS/DB store symbolic keys in either compressed or uncompressed format. (Note that CA IDMS/DB always strips trailing pad characters from an indexed set's symbolic keys.) If you specify compression, CA IDMS/DB applies a 2-level compression algorithm to the symbolic key before inserting the key into the index, as follows:

- **Prefix compression**—CA IDMS/DB compares (left to right) the symbolic key of the record being inserted into the index with adjacent symbolic keys and removes like characters. For example, if there are two symbolic keys, JOHNSON and JONES, CA IDMS/DB stores the JOHNSON key in its entirety and stores JONES as NES.
- **Repeating character compression**—CA IDMS/DB compresses three or more repeating single characters within each symbolic key into two bytes, and compresses 2 through 64 repeating blanks or nulls into one byte.

Specify compression of symbolic keys if the keys have either of the following characteristics:

- Commonly share the same prefix
- Contain many repeating characters (including blanks or nulls)

How the Index is Organized

To facilitate the process of locating an index entry for sorted sets, CA IDMS/DB organizes an index for sorted records into levels. In this case, when the first (top-level) SR8 record is full, CA IDMS/DB performs the following processing:

1. Splits the SR8 record into two parts. These two SR8 records stay at the same level.
2. Constructs a new higher level with two entries. Each entry points to one of the SR8 records created by Step 1.

CA IDMS/DB repeats this process as the index expands. Indexes can have any number of intermediate levels. As CA IDMS/DB adds new entries, it **splits** SR8 records and **spawns** new levels of SR8 records. An entry on one level points to an SR8 record at a lower level; the bottom-level entries point to the indexed database records themselves.

Therefore, in a sorted indexed set with three levels (top, intermediate, and bottom), the index is structured as follows:

- The **top level** is made up of one SR8 record that contains index entries. Each entry is composed of a pointer to (that is, the database key of) an intermediate-level SR8 record and the highest symbolic key contained therein.

- The **intermediate level** is made up of one SR8 record for each entry in the top level. Each entry is composed of a pointer to a bottom-level SR8 record and the highest symbolic key contained therein.

The **bottom level** is made up of one SR8 record for each entry in the intermediate level. Each entry is composed of a symbolic key and a pointer to a database record occurrence.

Example

For example, the sample database includes the indexed set EMP-LNAME-NDX. The EMP-LNAME-NDX set, shown in the following diagram and table shows the function of index levels and the search process. This simple index contains only three entries per SR8 record. The figure represents index and database records. (For simplicity, prior and owner pointers are not included in this figure.) The table shows the index pointers and symbolic keys.

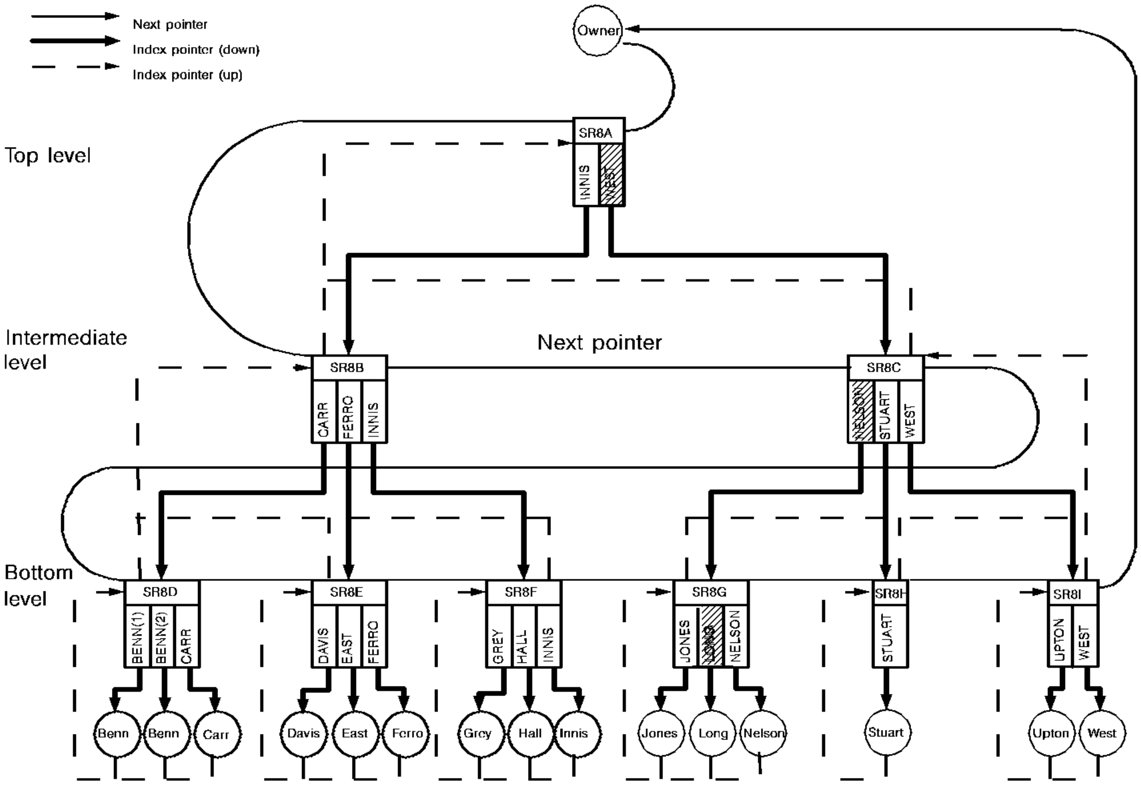
To locate LONG in this 3-level index, CA IDMS/DB performs the following steps:

1. Accesses the SR7 owner record by using the set name as the CALC key

   Note: For SQL-defined indexes, it uses a CALC key based on a number assigned to each index.

2. Accesses the top-level SR8 record by using the next pointer in the SR7 record
3. Searches this top-level SR8 record for the first entry with a symbolic key equal to or greater than LONG
4. Uses the db-key in this entry to access an intermediate-level SR8 record (that is, the NELSON/WEST SR8 record)
5. Searches this intermediate-level SR8 record (that is, the NELSON/WEST SR8 record) for the first entry equal to or greater than LONG
6. Uses the db-key in this entry to access a lower level SR8 record (that is, the JONES/NELSON SR8 record at the bottom level)
7. Searches this bottom-level SR8 record (the JONES/NELSON SR8 record) for the LONG entry
8. Uses the db-key in the LONG entry to access the requested member database record occurrence

Note: Since previous processing deleted indexed records, not all of the index entries in each SR8 record are presently used (for instance, the STUART and UPTON/WEST SR8 records at the bottom level). Consequently, this index has space for expansion without spawning a new level.



In this example, each SR8 record is composed of a maximum of three entries. Each entry is composed of a symbolic key value and a db-key. The shaded entries are used to locate the LONG record in the database. In the top and intermediate levels, the db-key in each entry points to another SR8 record. In the bottom level, the db-key in each entry points to a database record. (Note that, for simplicity, prior and owner pointers are not included in this figure; also, since two employees are named BENN, there are two database member occurrences with that name.)

The entries in the 3-level index are shown next. Each entry is composed of a symbolic key and a db-key. The shaded entries are used to locate the LONG record in the database. The index entries in the top and intermediate levels point to SR8 records at the next lowest level. Only the bottom-level entry points to the database record. Note that since two employees are named BENN, there are two db-keys (one to each database member occurrence) for that symbolic key.

SR8 db-key	SR8 Index Entries

	SR8 db-key	SR8 Index Entries	
Top level SR8 records	90002:3	Innis West	90004:10 90004:57
Intermediate level SR8 records	90004:10	Carr Ferro Innis	90015:13 90016:40 90030:6
	90004:57	Nelson Stuart West	90021:3 90018:53 90030:6
Bottom-level SR8 records	90015:13	Benn Carr	721009:147 723006:105
	90016:40	Davis East Ferro	720617:201 721592:63 722310:16
	90030:6	Grey Hall Innis	720016:31 727160:52 725921:74
	90021:3	James Long Nelson	726412:4 724263:12 727160:90
	90018:53	Stuart	720039:37
	90030:12	Upton West	720715:52 725129:2

Connecting Records to Indexed Sets

All set orders (that is, FIRST, LAST, NEXT, PRIOR, and SORTED) are supported for indexed sets. Indexed set order determines the way CA IDMS/DB builds the index when new member record occurrences are connected to the indexed set.

Connecting members to indexed sets ordered FIRST, LAST, NEXT, or PRIOR is discussed next, followed by a separate discussion of connecting members to indexed sets with a set order of SORTED.

Connecting Members to Unsorted Indexed Sets

To connect new members to indexed sets with FIRST, LAST, NEXT, and PRIOR set order, CA IDMS/DB inserts a new index entry between existing index entries. When one SR8 record fills, CA IDMS/DB creates a new SR8 record; there is only one level of SR8 records in an unsorted index.

Once a request has been made to connect a member occurrence to an indexed set, CA IDMS/DB first checks whether other entries exist. If no other entries exist, CA IDMS/DB creates and stores the first SR8 record (containing the first entry) and connects it to the owner occurrence with next, prior, and owner pointers. The target page for the first SR8 record is the page of the owner of the indexed set occurrence (plus displacement, if any).

CA IDMS/DB Actions

If other entries *do* exist, CA IDMS/DB takes the following actions:

Step 1

Identifies the appropriate SR8 record and insertion point based on the set order, as follows:

- **NEXT**—The insertion point is physically after the index entry for the current SET occurrence.
- **FIRST**—The insertion point is physically the first index entry in the first SR8 record.
- **PRIOR**—The insertion point is physically before the index entry for the current SET occurrence.
- **LAST**—The insertion point is physically the last index entry in the last SR8 record.

  Note: SQL-defined unsorted indexed constraints have an internal order of LAST.

Step 2

Inserts the new entry into the index, as follows:

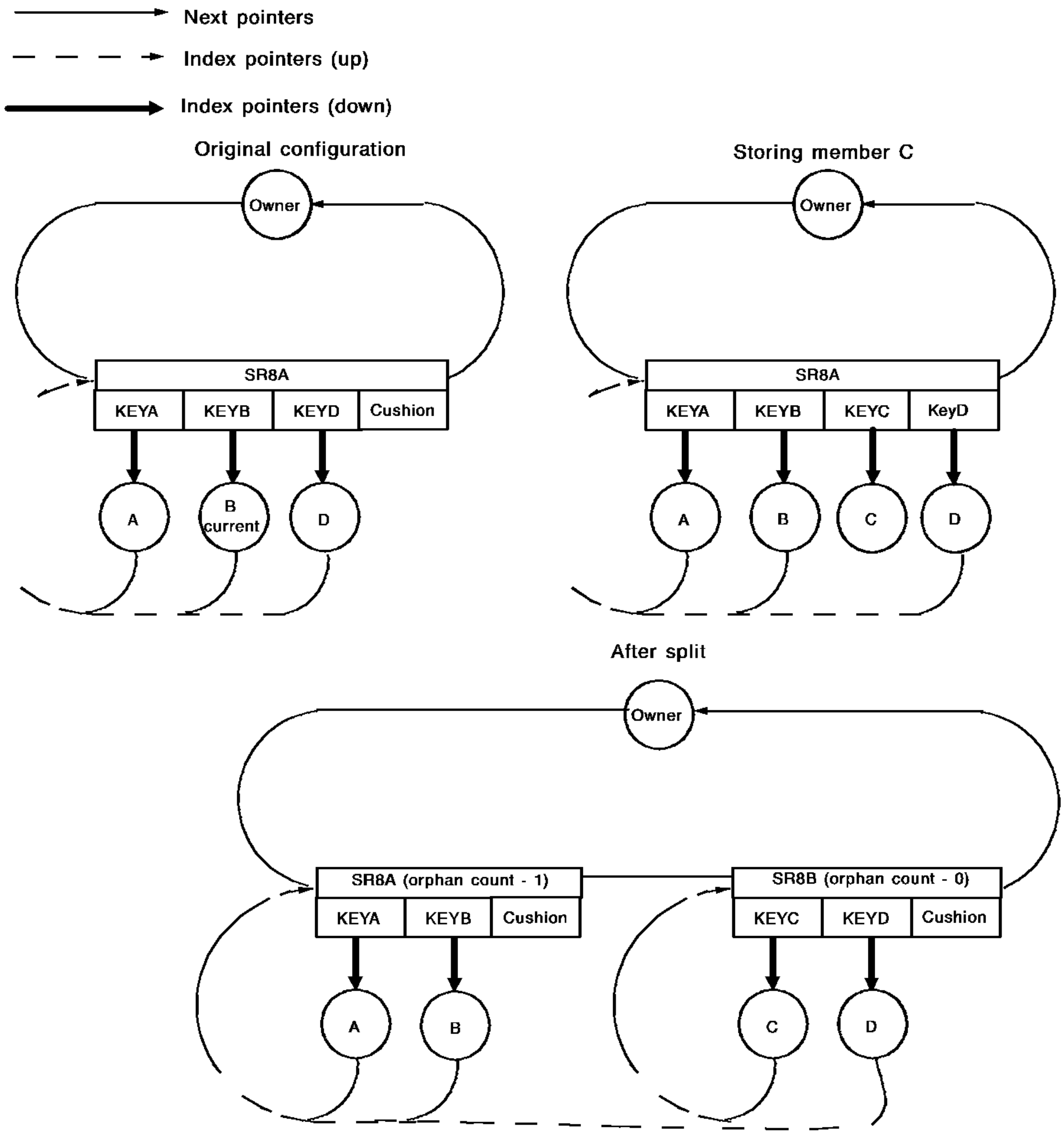
- If there is enough space in the target SR8 record for the new entry (that is, the insertion of this entry would not exceed the maximum allowable entries, and the target SR8's page has sufficient available space), CA IDMS/DB inserts the new entry into the target SR8 record.
- If there is *not* enough space in the target SR8 for the new entry, CA IDMS/DB inserts the new entry based on the location of the identified insertion point, as follows:
  - If the identified insertion point is physically first in the target SR8 record, CA IDMS/DB checks whether there is enough space in the prior SR8 record:
    - If there is enough space in the prior SR8 record, CA IDMS/DB inserts the new entry as the physically last entry in the prior SR8 record.
    - If there is not enough space in the prior SR8 record, CA IDMS/DB splits the target SR8 record.
  - If the insertion point is physically last in the target SR8 record, CA IDMS/DB checks whether there is enough space in the next SR8 record:
    - If there is enough space in the next SR8 record, CA IDMS/DB inserts the new entry in the next SR8 record.
    - If there is not enough space in the next SR8 record, CA IDMS/DB splits the target SR8 record.
  - If the insertion point is neither the physically first nor last in the target SR8 record, CA IDMS/DB checks whether there is enough space in the next SR8 record:
    - If there is enough space in the next SR8 record, CA IDMS/DB moves the last entry to the SR8 record
    - If there is not enough space in the next SR8 record, CA IDMS/DB splits the target SR8 record.

Index Pointers for Split SR8s

When CA IDMS/DB splits an SR8 record (Record A) into two SR8 records (Records A and B), the entries relocated to Record B point to member occurrences that still contain index pointers pointing to Record A if index pointers are maintained for the set (index pointers are optional for system-owned indexes). If index pointers are maintained, splitting Record A causes CA IDMS/DB to set the *orphan count* in Record A equal to the number of entries moved to Record B.

Splitting an SR8 Record

The following diagram shows splitting an SR8 record to add a member occurrence to an indexed set with a set order of NEXT. (Note that, for simplicity, prior and owner pointers are not included in this figure.)



Connecting Members to Sorted Indexed Sets

Spawning

CA IDMS/DB organizes an index for sorted records into levels. When a top or intermediate SR8 record is full, CA IDMS/DB **spawns** a new level through the following steps:

1. CA IDMS/DB splits the SR8 record into two SR8 records.
2. CA IDMS/DB constructs a new higher-level SR8 record. This new full-size SR8 record contains only two entries. Each entry points to one of the SR8 records created by Step 1.

   CA IDMS/DB determines the target page of a new SR8 record, as follows:

   - If displacement has been specified and if the new record is a bottom-level SR8 record, the target page is the page of the owner of the indexed set occurrence plus displacement.
   - Otherwise, the target page is the page of the owner of the indexed set occurrence.

CA IDMS/DB repeats this process as the index expands. Indexes can have any number of intermediate levels. As CA IDMS/DB adds new entries, it **splits** SR8 records and spawns new levels of SR8 records. An entry on one level points to an SR8 record at a lower level; the bottom-level entries point to the indexed database records themselves.

Connecting New Members

To connect new members into a sorted index, CA IDMS/DB first identifies the appropriate insertion point of the new entry based on the symbolic key or db-key. If this is the first entry (and, therefore, the first SR8 record), CA IDMS/DB creates, stores and connects a new SR8 record to the owner occurrence. CA IDMS/DB determines the target page for the new SR8 record as described above.

If this is not the first entry, CA IDMS/DB identifies the insertion point of the new entry based on the symbolic key or db-key. Once the appropriate insertion point is identified, CA IDMS/DB inserts the new entry into the index, as follows:

- If there is space in the target SR8 record (that is, if insertion of this entry would not exceed maximum allowable entries and the target SR8 record's page has sufficient available space, CA IDMS/DB inserts the new entry in the target SR8 record.
- If space in the target SR8 record is insufficient for the new entry, CA IDMS/DB attempts to move a number of entries to a prior or next SR8 record if space is available. Otherwise, a split occurs which may cause spawning depending on the available space in the higher-level SR8 records.

Disconnecting Records from Indexed Sets

Assume that the DEPT-EMPLOYEE set in the sample database is an indexed set sorted by employee identification number (EMP-ID-0415). To disconnect an occurrence of the EMPLOYEE record, a program could issue the following requests:

```
MOVE '0019' TO EMP-ID-0415.
FIND CALC EMPLOYEE.
DISCONNECT EMPLOYEE FROM DEPT-EMPLOYEE.
```

Processing the Request

CA IDMS/DB processes these requests as follows:

1. Finds the SR8 record pointed to by EMPLOYEE record 19's index pointer.
2. Searches the SR8 record for EMPLOYEE 19's db-key. If CA IDMS/DB finds EMPLOYEE 19's db-key, processing skips to Step 3. If CA IDMS/DB does *not* find EMPLOYEE 19's db-key, processing continues as follows:
   a. CA IDMS/DB decrements the SR8 record's orphan count by 1. If the SR8 contains no entries and the orphan count is 0 CA IDMS/DB erases the SR8 record.
   b. CA IDMS/DB follows SR8 records until it finds the db-key.
   c. CA IDMS/DB updates EMPLOYEE 19's index pointer to point to the correct SR8 record.
3. Removes EMPLOYEE 19's key entry from the bottom-level SR8 record and rewrites that SR8 record.
4. If this were an unsorted set, processing would be complete. Since this is a sorted set, if EMPLOYEE 19's symbolic key is the highest key in the SR8 record, CA IDMS/DB passes up the key to each level in which the key is the highest and removes the entry for EMPLOYEE 19 from each successive SR8 record.

Retrieving Indexed Records

In contrast to locating member records of a chained set, CA IDMS/DB locates member record occurrences in an index by searching the index. CA IDMS/DB does *not* have to access each member record occurrence as with chain linkage.

Types of Processing

Because CA IDMS/DB searches the index rather than actual record occurrences, indexed sets provide a quick and efficient method for the following types of processing:

- **Random retrieval by symbolic key or generic key**—CA IDMS/DB can retrieve individual records randomly by means of a symbolic key. CA IDMS/DB can also retrieve a group of records by using a partial (generic) symbolic key. A string of characters, up to the length of the symbolic key, can be used as a generic key.

- **Sorted retrieval by db-key, symbolic key, or generic key**—CA IDMS/DB can retrieve records in sorted order if the index is ordered on db-key or symbolic key. In this case, the db-keys or symbolic keys in an index are automatically maintained in sorted order and records therefore can be retrieved in ascending or descending order by db-key or symbolic key. Because records can be accessed through more than one index, they can be retrieved in more than one sort sequence.

- **Unsorted in exceptionally long sets**—To locate the db-keys of members in an indexed set, CA IDMS/DB walks the index. Since accessing member record occurrences' db-keys in an index requires less database I/O than accessing the record occurrences themselves, CA IDMS/DB can retrieve the db-keys of member records in exceptionally long sets more efficiently if the records are related using an index rather than a chain. This type of processing is useful for finding the *n*th record in a set or for manipulating lists of db-keys.

- **Physical sequential processing by db-key**—Member record occurrences can be clustered through an index. With this storage mode, the physical location of member records reflects the ascending or descending order of their db-key. If occurrences of a record type are to be retrieved in sequential order, clustering them through an index reduces I/O. This type of processing is most efficient when used with a stable database.

Example When Owner Pointers

Assume that the DEPT-EMPLOYEE set in the sample database is an indexed set sorted by employee identification number (EMP-ID-0415). To retrieve an occurrence of the EMPLOYEE record, a program might issue the following requests:

```
MOVE '0019' TO EMP-ID-0415.
OBTAIN CALC EMPLOYEE.
OBTAIN NEXT WITHIN DEPT-EMPLOYEE SET.
```

To fulfill the prior request, CA IDMS/DB performs the following processing:

Step 1

Using the value '0019' placed by the program in the EMP-ID-0415 field, obtains the EMPLOYEE record with an identification number of '0019'.

Step 2

Obtains the EMPLOYEE record with the next-highest identification number as follows:

1. Finds the SR8 record pointed to by EMPLOYEE 19's index pointer.
2. Searches the SR8 record for EMPLOYEE 19's db-key. If CA IDMS/DB finds EMPLOYEE 19's db-key, processing skips to Step 3. If CA IDMS/DB does *not* find EMPLOYEE 19's db-key, processing continues as follows:
   a. Decrements the SR8 record's orphan count by 1. If the orphan count is now 0, CA IDMS/DB erases the SR8 record if it is empty or rewrites it if it still contains entries.
   b. CA IDMS/DB searches the next SR8 record in the index until it finds the db-key. At this time, CA IDMS/DB updates the EMPLOYEE record's index pointer to point to the correct SR8 record.

      Note: CA IDMS/DB only updates pointers and the orphan count at this time if both the area that contains the SR8 records and the area that contains the EMPLOYEE records were readied in update mode.

Step 3

Obtains the EMPLOYEE record whose db-key is adjacent to current of set (that is, the next EMPLOYEE record).

Example When No Owner Pointers

If the EMPLOYEE record did not have owner pointers, a program could issue the following request to retrieve an occurrence of the DEPARTMENT record:

```
MOVE '0019' TO EMP-ID-0415.
OBTAIN CALC EMPLOYEE.
OBTAIN OWNER WITHIN DEPT-EMPLOYEE SET.
```

When fulfilling the above request, the DBMS would discover the lack of an owner pointer in the set and use the EMPLOYEE record's index pointer to find the bottom-level SR8 record that contains the key for the requested EMPLOYEE record. CA IDMS/DB will then use the owner pointer contained in that SR8 record to obtain the DEPARTMENT record.

SR8 Record Currency

When a program uses a subschema that contains records in an indexed set, CA IDMS/DB changes SR8 record currency **only when it accesses a member record through the index**, since CA IDMS/DB keeps track of SR8 record currency internally. When CA IDMS/DB accesses a member record in any other manner, CA IDMS/DB does *not* change SR8 record currency.

For example, a program might issue the following commands:

```
MOVE '0019' TO EMP-ID-0415.
OBTAIN CALC EMPLOYEE.
OBTAIN NEXT WITHIN EMP-LNAME-NDX SET.
```

CA IDMS/DB then fulfills these requests as follows:

1. Using the value '0019' for CALC entry into the database, CA IDMS/DB accesses the EMPLOYEE record with that identification number. EMPLOYEE 19 is now current of run unit, record, and the EMP-LNAME-NDX set. At this point, since CA IDMS/DB has not accessed an SR8 record, internal currency is *not* created for the SR8 structure of the index set.
2. On OBTAIN NEXT, CA IDMS/DB accesses the SR8 record that contains the index entry for EMPLOYEE 19. At this time, CA IDMS/DB makes this SR8 record the current record of the SR8 structure of the index set.
3. CA IDMS/DB finds the next index entry (either in that SR8 record or the next SR8 record). The SR8 record containing that index entry is now current of the SR8 structure.
4. Using the next index entry, CA IDMS/DB obtains the corresponding EMPLOYEE record. That EMPLOYEE record is now current of run unit, record, and the EMP-LNAME-NDX.

RETURN and FIND Commands

When a program uses a subschema that contains records in an index, use:

- **The RETURN command** to retrieve database keys and/or symbolic keys from the index without accessing database records.
- **The FIND command** to maintain indexed set currency.

Chapter 39: Lock Management

This section contains the following topics:

Controlling Access to CA IDMS Databases

Factors Controlling Access to Data

The primary means of influencing how CA IDMS/DB controls access to data is in the way areas are readied and the status assigned to areas within a central version. These factors ultimately determine which transactions can access and update data within a CA IDMS database and whether concurrent access is controlled at the area or record occurrence level.

The remainder of this section discusses:

- Readying areas
- The status of areas under the central version
- Default ready modes
- Ready modes and SQL access to data

Later sections in this chapter discuss how these factors determine the types of locks CA IDMS uses to control access to data.

Readying Areas

Area Ready Modes

Types of Ready Modes

A transaction can restrict runtime operations in a database area by readying that area with a mode of update or retrieval, as follows:

Type	Description
Update	The readying transaction can both retrieve and update data within the area.
Retrieval	The readying transaction cannot update data in the area.

Ready Mode Qualifiers

You can qualify the specified area ready mode with a shared (default), protected, or exclusive option to prevent update or retrieval of an area by other transactions executing concurrently under the same central version or, in the case of a shared area, under other central versions that are members of the same data sharing group. The qualified ready modes are:

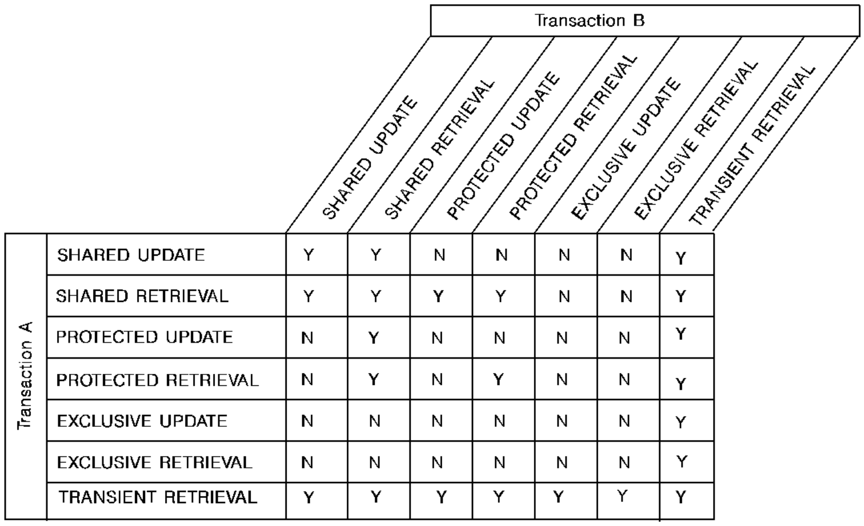
Ready Mode	Description
Shared update	If a transaction has readied the area in shared update mode, other transactions executing concurrently can ready the area in shared update or shared retrieval mode.
Shared retrieval	If a transaction has readied the area in shared retrieval mode, other transactions executing concurrently can ready the area in shared update, shared retrieval, protected retrieval, or protected update mode.
Protected update	If a transaction has readied the area in protected update mode, other transactions executing concurrently can ready the area in shared retrieval mode only.
Protected retrieval	If a transaction has readied the area in protected retrieval mode, other transactions executing concurrently can ready the area in shared retrieval or protected retrieval mode.
Exclusive update and exclusive retrieval	If a transaction has readied the area in exclusive update or exclusive retrieval mode, other transactions executing concurrently cannot ready the area in any mode. Exclusive retrieval is available only using navigational DML.

Ready Mode	Description
Transient retrieval	A ready mode of transient retrieval cannot be explicitly set by application programs or access module specifications. Instead, transient retrieval is automatically used by a transaction accessing an area in a retrieval mode, if either of the following conditions apply: ■ The status of the area within the central version is transient retrieval ■ The isolation level of the SQL transaction readying the area is transient read If a transaction has readied an area in transient retrieval mode, other transactions executing concurrently can ready the area in any mode.

Note: Both area status and transaction isolation levels area discussed under 39.2.2, "Central Version Area Status" and 39.2.4, “Ready Modes and SQL Access”, respectively.

Compatibility of Ready Modes

The mode in which one transaction readies an area restricts the mode in which other transactions executing under the same central version or in the case of a shared area, within the same data sharing group, can ready that area. This table shows the modes in which transaction B can ready an area, depending on the mode in which transaction A has readied the area. Y(es) signifies that the second transaction can ready the area in the specified mode; N(o) signifies that it cannot.



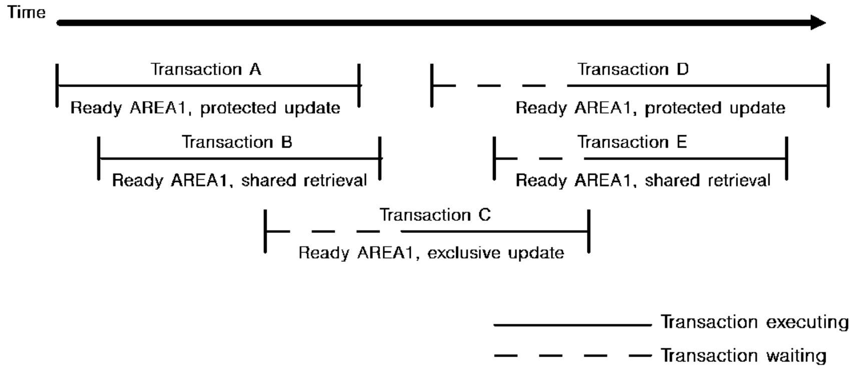
		Transaction B						
		SHARED UPDATE	SHARED RETRIEVAL	PROTECTED UPDATE	PROTECTED RETRIEVAL	EXCLUSIVE UPDATE	EXCLUSIVE RETRIEVAL	TRANSIENT RETRIEVAL
Transaction A	SHARED UPDATE	Y	Y	N	N	N	N	Y
	SHARED RETRIEVAL	Y	Y	Y	Y	N	N	Y
	PROTECTED UPDATE	N	Y	N	N	N	N	Y
	PROTECTED RETRIEVAL	N	Y	N	Y	N	N	Y
	EXCLUSIVE UPDATE	N	N	N	N	N	N	Y
	EXCLUSIVE RETRIEVAL	N	N	N	N	N	N	Y
	TRANSIENT RETRIEVAL	Y	Y	Y	Y	Y	Y	Y

Concurrent Use of an Area Within a Central Version or Data Sharing Group

When a transaction cannot ready an area because of a protected or exclusive restriction, CA IDMS/DB places the transaction in a wait state. When the restriction is lifted, the transaction can proceed.

Example of Concurrent Area Usage

The following diagram shows concurrent use of an area by transactions executing under a central version or data sharing group. Concurrently, transaction A readies AREA1 in protected update mode, transaction B readies the area in shared retrieval mode, and transaction C attempts to ready the area in exclusive update mode. The system puts transaction C into a wait state until both transactions A and B terminate. Transactions D and E, attempting to ready the area, must wait until transaction C terminates.


Time
Transaction A
Ready AREA1, protected update
Transaction B
Ready AREA1, shared retrieval
Transaction C
Ready AREA1, exclusive update
Transaction D
Ready AREA1, protected update
Transaction E
Ready AREA1, shared retrieval
Transaction executing
Transaction waiting

Central Version Area Status

Area Status and Ready Modes

Each area accessible from within a central version has a status associated with it. The status of an area affects the mode in which transactions executing under the central version can ready the area:

Mode	Description
UPDATE (or ONLINE)	Transactions executing under the central version can ready the area in any mode
RETRIEVAL	Transactions executing under the central version can ready the area in any retrieval mode (EXCLUSIVE, PROTECTED, SHARED or TRANSIENT)
TRANSIENT RETRIEVAL	Transactions executing under the central version can ready the area in any retrieval mode, but the CA IDMS/DB automatically changes the mode to TRANSIENT RETRIEVAL

Mode	Description
OFFLINE	Transactions executing under the central version cannot ready the area in any mode

Establishing the Area Status

The status of an area within a central version is initially established by specifications made within the DMCL used by the DC/UCF system. An area's status may subsequently be changed by DCMT commands.

Permanent Area Status

When an area's status is changed through a DCMT command, it may be designated as permanent. A permanent area status persists across both normal and abnormal system terminations until it is subsequently changed by another DCMT command or until the journal files associated with the central version are initialized. Whether an area's status has been designated as permanent is indicated on the output from a DCMT DISPLAY AREA command.

At System Startup

If a permanent area status is not in effect, the first time a system is started and each time it is subsequently started after a normal shutdown, the status of the area is set to that specified in the ON STARTUP parameter of the ADD SEGMENT or ADD AREA statement within the DMCL definition. The default area status is UPDATE.

Following an Abnormal System Termination

If a permanent area status is not in effect, when restarting a system following an abnormal termination, the status of the area is set to that specified in the ON WARMSTART parameter of the ADD SEGMENT, or ADD AREA statement within the DMCL definition. The area can be set to what it was at the time of the failure (the default) or it can be set to an explicit value.

Changing Area Status

You can change the status of an area within a central version by issuing a DCMT VARY AREA or VARY SEGMENT command. In certain cases, CA IDMS cannot change the status of the area immediately because existing transactions are accessing the area. In addition to active transactions, longterm or notify locks held by pseudo-conversational applications may prevent the area status from being changed. If CA IDMS cannot change the status immediately, it initiates an internal task that completes the DCMT VARY operation when no more conflicts exist. During the time it takes to complete the vary, transactions attempting to ready the area in a mode that is incompatible with the new area status receive an error.

Note: For more information about the DCMT VARY AREA and VARY SEGMENT commands, see the *CA IDMS System Tasks and Operator Commands Guide*.

Default Ready Mode Using Navigational DML

You can specify a *default ready mode for a database area* in a subschema definition. The specified default mode determines the mode in which the area is to be readied for programs using that subschema. If you specify a default mode for a database area, programs using the subschema do *not* have to issue a READY command for the area. If a program issues a READY command for one area in the subschema, the automatic readying mechanism is disabled. In this case the program must issue a READY command for all areas to be accessed unless the FORCE option is specified for the default usage mode. Areas using the default usage mode combined with the FORCE option are automatically readied even if the run-unit already issued READY for other areas.

Note: For more information on the limitations of using the FORCE option with ADS dialogs, see the Usage (see page 466) information in the Area Statement section.

Ready Modes and SQL Access

Factors Affecting SQL Lock Management

When accessing data using SQL, the way in which an area is readied depends on several factors:

- The transaction state
- The isolation level
- The requested ready mode
- The status of areas within central version

Transaction State

A transaction initiated using SQL has one of two states:

State	Description
READ ONLY	Data can be read, but not updated; updates to temporary tables *are* allowed
READ WRITE	Data can be both read and updated using DML and DDL statements

Default is READ WRITE

Unless otherwise specified, the transaction state is READ WRITE. You can override the default when you define an access module or by issuing a SET TRANSACTION statement at runtime.

Isolation Level

A transaction initiated using SQL also has one of two isolation levels:

Isolation Level	Description
CURSOR STABILITY	Guarantees read integrity. Read integrity ensures that: ■ All data accessed by the transaction is in a committed state ■ The most-recently accessed row of an updatable cursor is protected from update by other transactions while it remains current
TRANSIENT READ	Does not guarantee read integrity. For this reason, a transaction executing under transient read is not allowed to update the database. If the isolation level of a transaction is transient read, the transaction state is automatically READ ONLY.

Default is CURSOR STABILITY

Unless otherwise specified, the isolation level of a transaction is CURSOR STABILITY. You can override the default when you define an access module or by issuing a SET TRANSACTION statement at runtime.

Requested Ready Modes

You can specify within the access module definition the modes in which CA IDMS/DB is to ready the areas accessed by non-dynamic SQL statements embedded in an application. Otherwise, CA IDMS/DB determines the ready mode at runtime. It also determines the ready mode at runtime for dynamic SQL statements.

Runtime Ready Modes

The ready mode in which an area is accessed at runtime depends on the requested ready mode, the transaction state, the isolation level, and the area's availability:

Transaction State	Isolation Level	Area Ready Mode
READ ONLY	TRANSIENT READ	Transient retrieval mode; no row locks are placed.
READ ONLY	CURSOR STABILITY	Retrieval modes only. If update modes were specified on the CREATE or ALTER ACCESS MODULE statement, CA IDMS/DB changes them to shared retrieval. If no ready option was specified, the default is shared retrieval.
READ WRITE	CURSOR STABILITY	All areas are accessed using the mode specified on the CREATE ACCESS MODULE. If no mode was specified, the default is: ■ Shared update in local mode and under the central version, if the area status is update ■ Shared retrieval under the central version, if the area status is retrieval

Under central version, if an area is being readied in a retrieval mode and the status of the area is transient retrieval, CA IDMS/DB changes the ready mode to *transient retrieval*.

Physical Area Locks

CA IDMS/DB maintains a physical area lock as a flag within the first space management page of an area. It examines and sets the lock whenever the area is opened for update. This occurs when:

- A local mode transaction readies the area in an update mode
- A DC/UCF system is started in which the area status in the DMCL is UPDATE
- A DCMT VARY AREA command changes the status of the area to UPDATE

Unlocking the Physical Area

Once a physical lock is placed on an area, it remains set until:

- The local mode transaction or central version terminates normally
- Manual recovery procedures are used to roll out the effects of a failing local mode transaction
- A central version is restarted after an abnormal termination and subsequently shutdown normally
- The area status within central version is changed from update to another status

Physical Area Locks and Shared Areas

In a data sharing environment, the physical area lock in a shared area is set by the first member of a data sharing group to open the area for update and it is reset by the last member to relinquish control.

Controlling Update Access

Purpose of Physical Area Locks

Physical area locks prevent concurrent update by independent transactions (that is transactions executing outside of a single central version or data sharing group) and prevent update access to an area requiring recovery of incomplete transactions.

How Locking Works

CA IDMS/DB provides this protection as follows:

Protection Name	Description
Local mode	As each area is readied in any update mode, CA IDMS/DB checks the lock. If the lock is set, the transaction receives an error and access to the area is not allowed. If the lock is not set, the local mode transaction causes the lock to be set and the space management page is rewritten immediately. If the transaction terminates abnormally (that is, without issuing a FINISH or COMMIT WORK), the lock remains set. Further update access is prevented until the area is recovered (through CA IDMS recovery procedures).
Central version	At system startup, the central version checks the locks in all areas intended for update. If the physical lock is set and the area is not shared, or the area is shared but is not currently being updated by another member of the central version's data sharing group, then a warning message is displayed at the console and the area status is changed to offline. The central version proceeds without the use of that area and any transaction attempting to ready that area will receive an error. If the physical lock is subsequently removed from the area, the status of the area can be varied to update.

Locking Within Central Version

Logical Locks

Control Access to Resources

Logical locks are used within a central version and within a data sharing group to control access to database resources by concurrently executing transactions. Before a transaction can access a resource, it places a lock on the resource which prevents other transactions from modifying or, in some cases, accessing the resource while the lock is maintained.

In a data sharing environment, CA IDMS uses global transaction locks, maintained in a coupling facility lock structure to control inter-member access to shared resources.

Note: For more information about global locking, see 39.5, “Locking Within a Data Sharing Group”.

Wait States

If a transaction attempts to lock a resource which is locked by another transaction with a conflicting mode, the first transaction will wait until the lock is released. If the waiting transaction exceeds the internal wait interval specified at system generation, CA IDMS aborts the transaction and rolls out its updates. If one transaction is waiting to place a lock and the transaction that holds it then waits on a lock held by the first transaction, a deadlock condition exists. CA IDMS resolves this condition by aborting and rolling back one of the transactions.

Note: For more information about detecting and resolving deadlocks, see 39.7, "Deadlocks".

Hierarchical Locking

CA IDMS/DB uses a hierarchical locking scheme in which locks are placed at area and record occurrence (row) levels. Area locks control access to the area, and by implication, all record occurrences stored within the area. Record locks control access to individual record occurrences or rows.

A transaction intending to access data within an area must first place a lock at the area level. Depending on the strength of that lock (its mode), the transaction may or may not also place locks on individual record occurrences as it retrieves or updates them.

In a data sharing environment, CA IDMS uses a three-level hierarchy to control inter-member access to shared resources: area, proxy, and record occurrence.

Note: For more information about this additional level, see 39.5.3, “Proxy Locks”.

Types of Locks

Lock Modes

Each logical lock has an associated lock mode. The mode of the lock determines whether the lock conflicts with other locks already held on the resource and with locks subsequently requested by other transactions.

The following types of locks (lock modes) are used for both area and record locks:

Mode	Identifier	Description
Share	S	Typically used to guarantee that no updates are made to data while a transaction is accessing it. A share lock is compatible with other share locks but not with exclusive locks. A share lock placed on an area implies a share lock on each record within the area.
Exclusive	X	Typically placed on a resource to protect transactions from accessing data that is being updated by the issuing transaction. An exclusive lock is incompatible with both share and other exclusive locks. An exclusive lock placed on an area implies an exclusive lock on all records within the area.
Null-lock	NL	A null-lock is a special type of lock which is placed on a record to signify a notify lock and on an area to signify transient retrieval access. Null-locks provide no protection against concurrent access.

Intent Locks on Areas

The following types of locks are placed only on areas:

Mode	Identifier	Description
Intent share	IS	Allows share (S) locks to be placed on records within the area.
Intent exclusive	IX	Allows exclusive (X) locks to be placed on records within the area.
Update intent exclusive	UIX	Allows exclusive locks to be placed on records within the area by the issuing transaction, but not by other transactions.

In addition to the above lock modes, the following lock mode has been provided for but is currently not used:

- Update (U)---An update lock is placed on a resource if it might be updated after it is retrieved (in which case the lock would be upgraded to an exclusive lock).

Compatibility of Locks

For two transactions running within the same DC/UCF system to access the same area or row concurrently, their lock types must be compatible. When two transactions attempt to set locks that are not compatible, the first transaction to set a lock causes the second transaction to wait until the resource is freed.

Note: CA IDMS ensures that a transaction does not compete with itself for locks.

Compatibility Chart

The following chart shows which lock modes are compatible and which are incompatible. The plus sign (+) indicates a situation in which two lock modes are compatible. The minus sign (-) indicates a situation in which two lock modes are incompatible.

	NL	IS	IX	S	U	UIX	X
NL	+	+	+	+	+	+	+
IS	+	+	+	+	+	+	-
IX	+	+	+	-	-	-	-
S	+	+	-	+	+	-	-
U	+	+	-	+	-	-	-
UIX	+	+	-	-	-	-	-
X	+	-	-	-	-	-	-

Example

If TRANSACTION1 holds a share (S) lock on an area, TRANSACTION2 can set a null-lock (NL), intent-share (IS), share (S), or update (U) lock on the same area.

Logical Area Locks

Effect of Ready Mode

To control concurrent access to areas within a central version, the mode in which an area is readied is translated into a logical lock on the area. As an area is readied, CA IDMS/DB attempts to place an appropriate lock based on the ready mode. If the new lock doesn't conflict with locks already held by other transactions, access is granted to the area. If a conflict exists, the transaction is placed in a wait state until the conflicting locks are released.

Area Lock Depends on Area Ready Mode

The type of lock (lock mode) placed on an area depends on the mode in which the area is being readied:

Ready mode	Lock mode
Transient retrieval	Intent Share (NL)
Shared retrieval	Intent Share (IS)
Shared update	Intent Exclusive (IX)
Protected retrieval	Share (S)
Protected update	Update Intent Exclusive (UIX)
Exclusive retrieval	Exclusive (X)
Exclusive update	Exclusive (X)

When Area Locks are Acquired

For transactions initiated through navigational DML, CA IDMS/DB acquires area locks when either of the following occur:

- The first non-ready DML (other than BIND RECORD) statement is issued following one or more READY statements
- The first non-bind statement is issued within a transaction using default ready modes specified by the subschema

For SQL-initiated transactions, when area locks are acquired depends on the area acquisition mode specified within an access module or in effect for dynamic SQL.

Note: For more information, see 39.4.4, “Area Locking for SQL Transactions".

Area Acquisition Threshold

If a transaction is locking multiple areas at one time, and must wait to place a lock on one of the areas, CA IDMS/DB releases the locks on all other areas before placing the transaction in a wait state. This helps to avoid deadlocks between two or more transactions trying to gain access to areas. However, it also means that another transaction can gain access to an area whose lock was released by the waiting transaction. To avoid this pre-emption, you can specify an area acquisition threshold at system generation that limits the number of times a transaction will wait on an area lock before it no longer releases other area locks.

Area Locking for SQL Transactions

When Area Locks are Acquired

The time at which area locks are acquired for SQL transactions varies depending on the lock acquisition mode in effect. There are two lock acquisition modes:

- Preclaim
- Incremental

On First Database Access

The preclaim mode directs CA IDMS to place locks on all areas in a transaction that use the preclaim acquisition mode as soon as the first statement that requires access to the *database* is executed.

You can use the preclaim mode to reduce the likelihood of deadlocks. A transaction that uses the preclaim option to lock an area will not wait for an area that is held by another transaction while it holds a lock on an area.

On First Area Access

The incremental mode directs CA IDMS to delay placing a lock on an area until the first statement in the transaction that requires access to the *area* is executed.

You can use the incremental mode to increase database concurrency. A transaction that uses the incremental mode does not place a lock on an area until the area is actually required for processing. This makes the area accessible to other transactions for a longer period of time. In general, if a transaction does not always access every area in its access path, you should assign the incremental mode to those areas that are least likely to be accessed.

Example

Suppose a transaction needs to access three different tables, each of which is stored in a different area:

Table	Area	Acquisition mode
T1	AREA1	Preclaim
T2	AREA2	Incremental
T3	AREA3	Preclaim

Locks would be acquired in the manner shown next:

```
TRANSACTION A
-------------
   .
   .
   .
SELECT * FROM T1;    ◄--------------  Locks are placed on both
   .                                    AREA1 and AREA3.
   .
   .
SELECT * FROM T2;    ◄--------------  A lock is placed on AREA2.
   .
   .
   .
SELECT * FROM T3;
   .
   .
   .
```

Record Locks

Purpose of Record Locks

Record locks are used within the central version to control concurrent access to individual record occurrences (rows). Occurrence-level record locks (in conjunction with area locks) are used to:

- Protect against concurrent update of the same record by two or more transactions
- Prevent record occurrences that are current within one transaction from being updated by another transaction

Implicit record locks

CA IDMS/DB automatically places locks on records accessed by a transaction if the area in which the record resides is readied in any of the following modes:

Area ready mode	Record lock
Shared retrieval on read records	Shared (S) locks
Shared update	Shared (S) locks on read records; exclusive (X) locks on updated records
Protected update	Exclusive (X) locks on updated records

Note: You can use system generation options to inhibit record locking for navigational DML applications, as discussed in 39.4.6, “System Generation Options Affecting Record Locking".

Shared Record Locks

If shared locks are being maintained, CA IDMS/DB places one on each record as it is accessed. Shared locks are also maintained on:

- The most-recently accessed record of its type (the most-recently accessed row of each table)
- The most-recently accessed record in each set (the most-recently accessed row of each constraint or index)
- The most-recently accessed record in each area.

Note: Additional shared locks are maintained on the current row of each updatable cursor open within an SQL transaction.

CA IDMS/DB releases these locks as the transaction accesses different record occurrences. These implicit record locks guarantee the integrity of the currencies used by navigational DML applications and provide the protection necessary for SQL applications executing with an isolation level of cursor stability.

Note: For more information about isolation levels, see 39.2.4, "Ready Modes and SQL Access".

Exclusive Record Locks

If exclusive locks are being maintained, CA IDMS/DB places them on all records altered by a DML or DDL statement until the recovery unit terminates (that is a COMMIT (CONTINUE), ROLLBACK (CONTINUE) or FINISH is issued) or until the transaction abends.

Implicit Page Locks

Implicit locks are used in a special way to control user access to pages for which the amount of available space has been altered. When the available space on a page is changed as a result of an update operation, CA IDMS/DB places a special implicit exclusive lock on the page, allowing retrieval to continue. If a subsequent DML or DDL command from a different transaction requests further modification to available space on that page, the request is delayed until the lock is released (that is, until the recovery unit that caused the lock to be set terminates).

Explicit Record Locks

The navigational programmer can set *explicit record locks* with the DML KEEP command. The KEEP verb or the KEEP option of a FIND or OBTAIN verb places a shared lock on the record occurrence. KEEP with the EXCLUSIVE option places an exclusive lock on the record occurrence. CA IDMS/DB holds explicit record locks until the transaction terminates or a COMMIT ALL statement is executed.

System Generation Options Affecting Record Locking

Two system generation options affect whether CA IDMS/DB maintains record locks for navigational DML transactions. These options are:

Option	Description
RETRIEVAL LOCK/NOLOCK	Specifies whether CA IDMS/DB places shared locks on records in an area readied in SHARED RETRIEVAL
UPDATE LOCK/NOLOCK	Specifies whether CA IDMS/DB places exclusive locks on records updated in an area readied in PROTECTED UPDATE

Note: These system generation parameters affect only navigational DML applications; they do not apply to SQL applications.

Reading Uncommitted Data

If RETRIEVAL NOLOCK is specified, a transaction may read uncommitted data; that is, it may read data that has been updated by another transaction before those changes have been committed or data that has been accessed by a retrieval transaction may be concurrently updated while the retrieval transaction is still active. This may result in inconsistencies in the data processed by the shared retrieval transaction. These inconsistencies may also include transient 11xx abends from the DBMS.

If UPDATE NOLOCK is specified, a transaction updating data in an area readied in PROTECTED UPDATE does not protect transactions readying the area in SHARED RETRIEVAL. As with RETRIEVAL NOLOCK, it is possible for a transaction which has readied the area in SHARED RETRIEVAL to read a record updated by a PROTECTED UPDATE transaction before it has been committed.

Since both options affect the protection afforded shared retrieval transactions, it is typical (though not required) to set both parameters in the same way. In systems in which there is a high volume of updates, you might want to consider specifying LOCK for both.

Note: No inter-CV retrieval protection is provided except for shared areas accessed through members of a data sharing group. If an area is not shared, then regardless of the system lock options in effect, it is possible for a shared retrieval transaction executing in a central version whose area status is retrieval to read uncommitted data updated by another central version.

TRANSIENT RETRIEVAL area status

As an alternative to using system generation parameters to reduce the volume of record locks maintained, consider using a central version's area status of TRANSIENT RETRIEVAL instead. Provided the area is not updated within the central version, a status of transient retrieval can be used to eliminate the locking of records within the area.

Locking Within a Data Sharing Group

Within a data sharing group, locking is used to control inter-member access to shared resources, just as it is used to control access to resources within a central version. The basic locking scheme used within a central version is extended for data sharing in the following ways:

- Global transaction locks are used to control inter-member access
- An additional level is introduced in the locking hierarchy
- Page locks are used to protect database pages while they reside in a buffer pool

Note: For more information about data sharing, see the *CA IDMS System Operations Guide*.

Inter-CV-Interest

Inter-CV-interest denotes a state in which an area is being shared by:

- At least one group member with an area status of UPDATE, and
- More than one group member with an area status of RETRIEVAL or UPDATE. Members accessing an area in TRANSIENT RETRIEVAL, have no impact on inter-CV-interest.

Conflict for the area (and the records and pages in the area) can only occur if there is inter-CV-interest in the area. This is significant because if there is no inter-CV-interest in an area, the overhead associated with controlling access to it is reduced.

Whether there is inter-CV-interest in an area is indicated on the output from a DCMT DISPLAY AREA command.

Global Transaction Locks

Global transaction locks are locks that reside within a coupling facility lock structure and are used to control inter-member access to data in shared areas. Whenever a transaction places a lock on a shared area or on a record that resides in a shared area and there is inter-CV-interest in that area, global locks ensure that no other transaction in the data sharing group is accessing the same resource in a conflicting mode.

Managing Global Locks

Global locking relies on a coupling facility lock structure to record and manage global locks. Global locks are acquired by the CA IDMS lock manager whenever a transaction places a lock on a resource and a sufficiently strong global lock is not already held by that CV. Global locks are retained until no transaction within a CV requires a lock of that strength, at which point the global lock may be released, downgraded, or retained, depending on the resource type and whether there is contention for the resource between group members.

Inter-CV-Interest and Global Locking

Global transaction locks are not acquired if there is no inter-CV-interest in an area. If inter-CV-interest begins because another member accesses the area in a potentially conflicting mode, global transaction locks will be acquired by every sharing member in which a transaction holds a lock on the area or any of its records.

Proxy Locks

A proxy lock is a global lock used within a data sharing group to represent a lock on all the records within a page of a shared area. Proxy locks are held by members of a data sharing group and not by individual transactions.

An Additional Hierarchy Level

Proxy locks represent an additional level in the locking hierarchy used by CA IDMS to control access to data.

Normally CA IDMS uses a two-level locking hierarchy: area and record. Before placing a lock on a record, a transaction must place a lock on the area in which the record resides. Depending on the mode of the area lock, it may be possible to avoid placing locks on individual records within the area.

For shared areas, the locking hierarchy expands to three levels: area, proxy, and record. Before a lock is placed on a record in a shared area, a lock must be held on a proxy that represents the record's page and before this can be done, a lock must be held on the area in which the record resides.

Proxy Lock Modes

A proxy can be locked in one of two modes: Share or Exclusive. At least a share lock must be held on a proxy before a transaction can place a share or null (notify) lock on a record represented by the proxy. Similarly, an exclusive proxy lock must be held before a transaction can place an exclusive lock on a record represented by the proxy.

Managing Proxy Locks

An exclusive proxy lock held by one member does not prohibit access by another member. Instead the purpose of proxy locks is to detect inter-CV contention for resources and to eliminate the use of global record locks where possible. As long as all members holding a lock on a proxy hold it in share mode, there is no contention for resources on the page and no need to globally lock individual records on that page. However, if at least two members hold a lock on a proxy and at least one of those is an exclusive lock, then there is possible contention for individual records, necessitating the use of global record locks to control access to individual records.

The acquisition and management of proxy locks is done automatically by the CA IDMS lock manager. Application programs do not need to explicitly acquire or manage proxy locks. However, database administrators should be aware of their existence and their impact on recovery and resource utilization.

Page Locks

A page lock is a lock that is used within a data sharing group to protect database pages while they reside in a member's local buffer pool. Page locks are only placed on pages of areas that are designated for data sharing and only if there is inter-CV interest in the area.

Managing Page Locks

The coupling facility lock structure associated with the data sharing group is used to record and manage global page locks, just as is done for global transaction locks. And just as a proxy represents all of the records on the page, it also represents the page itself. Therefore, proxy locks reduce the need to acquire and release global page locks each time a page is moved into and out of the buffer pool.

Page Lock Protection

Before a database page is read into the buffer pool, an exclusive or shared lock is placed on that page, depending on whether the active transaction intends to update the page. Once the lock is acquired, no other group member may place a conflicting lock on the page until the first member relinquishes its lock. This means that no other sharing member may read the page contents while another member has it locked exclusively. Page locks are held until another group member wants access to the page in a conflicting mode. Before an exclusive page lock can be released on an updated page, the page is written to the disk and to the shared cache.

Controlling Access to Native VSAM Files

Physical Area Locks Not Set

CA IDMS does not maintain physical area locks for areas that map to native VSAM data sets. Therefore, a combination of SHAREOPTIONS, JCL, and operational procedures is used to control updates of native VSAM data sets by CA IDMS, local transactions, and non-CA IDMS programs.

DEFINE CLUSTER Command

For example, you can prevent concurrent update by specifying the parameter SHAREOPTIONS(2,3) in the DEFINE CLUSTER command during VSAM cluster definition. This parameter permits only one application program to open the data set for update, thereby preventing concurrent update of the data set by two application programs executing in different regions. Within a central version, access to native VSAM files is controlled through the use of logical locks on areas and records just as for CA IDMS database files.

CA IDMS/DB Facilities

You can use CA IDMS facilities to further control access to native VSAM data sets. For example, *to protect a data set from being updated*, set the status within central version to TRANSIENT RETRIEVAL or RETRIEVAL. In this case, no application program running under the central version can ready the area in update mode.

To ensure read integrity of the area when accessed by transactions executing under a central version, you can use the following procedure when updating the data set using non CA IDMS programs:

1. Vary offline the CA IDMS area that maps to the VSAM data set by means of the DCMT VARY AREA OFFLINE command.
2. Run the job to update the VSAM data set.
3. Vary the area to retrieval access mode using the DCMT VARY AREA RETRIEVAL command, thus making the area once again available under the central version.

Deadlocks

A *deadlock* is an unresolvable contention between multiple requestors for a resource. Resources are either DC/UCF system resources (such as programs and storage) or database resources (such as areas and records). CA IDMS/DB uses different control block structures to track contention for DC/UCF system resources and database resources. Although it tracks deadlocks using different control block structures, the same deadlock detection mechanism is used to resolve deadlocks.

How the System Detects a Deadlock

Deadlock detection is a process performed on a time interval basis. It is carried out in four major phases:

1. **Identifying stalled tasks**—To identify tasks that are stalled, all dispatch control elements (DCEs) in the system are examined. Any DCE found stalled while waiting on an internal resource is entered into the deadlock detection matrix (DDM). All subsequent processing begins with the DCE address stored in the DDM table. This eliminates the need to scan all DCEs in the system.
2. **Identifying task dependencies**—Next, the dependencies between the stalled tasks are identified. The deadlock detection matrix is updated. For each task on which another task is waiting, a bit in the deadlock detection matrix is set to one.
3. **Identifying deadlocks**—To determine which tasks are involved in a deadlock cycle, a transformation is performed on the matrix. From this process, a pair of deadlocked tasks is identified. From this pair, a victim is selected.

4. **Selecting a victim**—The task running for the shortest period of time is chosen as the victim of the two tasks as long as:

   - The priority of the victim task is less than that of the other task
   - The victim task's wait was not entered with COND=NONE and the other task's wait was entered with COND=DEAD

   The task running for the shortest period of time is chosen as the victim because it is more likely that it will have consumed fewer resources than a longer running task. As a result, less duplication of work should be required when the victim is restarted, with these exceptions:

   - If the other task is of a higher priority, implying that it is of more importance
   - If the victim task entered the deadlock with COND=NONE and the other task specified COND=DEAD. In this case, the task specifying COND=DEAD is chosen as the victim since COND=DEAD indicates that the task is designed to handle and recover from deadlock situations. This prevents an abend.

Victim Selection User Exit

The algorithm used to select a victim in a deadlock situation may not be optimal for your installation or applications. User exit 30 allows victims to be selected based upon specific requirements. The exit is passed the DCE addresses of each pair of deadlocked tasks and may take one of two actions:

- Choose one of the tasks as the victim task
- Return control to the deadlock detector by requesting that the default deadlock detection logic be applied

Note: For a discussion of user exit 30, see the *CA IDMS System Operations Guide*.

Deadlock Detection Interval

You can control the frequency with which the deadlock detection mechanism searches for deadlocked tasks using the DEADLOCK DETECTION parameter of the SYSTEM statement.

The DEADLOCK DETECTION parameter allows you to specify the amount of time that elapses before the deadlock detection mechanism searches for deadlocked tasks. Note that in an idle system, deadlock detection is also idled until new tasks are started. This eliminates CPU consumption for deadlock detection when no tasks could possibly be deadlocked.

You can use the DCMT VARY DEADLOCK command at runtime to override the system generation specification.

More Information

- For more information about the DEADLOCK DETECTION parameter of the SYSTEM statement, see the *CA IDMS System Generation Guide*.
- For more information about the DCMT VARY DEADLOCK command, see the *CA IDMS System Tasks and Operator Commands Guide*.

Global Deadlock Detection

What is a Global Deadlock?

A global deadlock is a situation in which unresolvable contention exists for shared resources between tasks executing on different members within a data sharing group.

Detecting Global Deadlocks

A global deadlock is possible if at least one stalled task is waiting on a global resource. In a potential global deadlock situation, each member passes information to the one acting as the global deadlock manager. The global deadlock manager examines the information gathered from the other members and determines which tasks, if any, are deadlocked.

Resolving Global Deadlocks

If a global deadlock exists, user exits are invoked, to assist in selecting a victim task. If these exits are not provided, the task running for the shortest period or with the lowest priority is designated as the victim. Once the victim is determined, the member on which the victim is executing is directed to cancel the task.

Global Deadlock User Exits

Two user exits are used in selecting a victim in a global deadlock situation:

- **Exit #35** is invoked when a group member is collecting information about a stalled task to send it to the global deadlock manager. The exit provides the opportunity for a site to collect additional information that may be relevant to the victim selection process.
- **Exit #36** is invoked by the global deadlock manager when a victim is being selected. Its function is similar to exit #30, but it is passed different parameters. Instead of being passed the DCE addresses of two deadlocked tasks, it is passed a pair of parameters for each task, one of which is the information collected by exit #35. In this way, site-specific criteria can be used in selecting a victim even though the deadlocked tasks may be executing on a group member that is different than that of the global deadlock manager.

Note: For details on coding these exits, see the *CA IDMS System Operations Guide*.

Appendix A: Sample SQL Database Definition

Sample Database Definition

```
 CREATE SCHEMA DEMOEMPL;
 SET SESSION CURRENT SCHEMA DEMOEMPL;

CREATE TABLE          BENEFITS
    (FISCAL_YEAR          UNSIGNED NUMERIC(4,0)                NOT NULL,
     EMP_ID               UNSIGNED NUMERIC(4,0)                NOT NULL,
     VAC_ACCRUED          UNSIGNED DECIMAL(6,2)   NOT NULL WITH DEFAULT,
     VAC_TAKEN            UNSIGNED DECIMAL(6,2)   NOT NULL WITH DEFAULT,
     SICK_ACCRUED         UNSIGNED DECIMAL(6,2)   NOT NULL WITH DEFAULT,
     SICK_TAKEN           UNSIGNED DECIMAL(6,2)   NOT NULL WITH DEFAULT,
     STOCK_PERCENT        UNSIGNED DECIMAL(6,3)   NOT NULL WITH DEFAULT,
     STOCK_AMOUNT         UNSIGNED DECIMAL(10,2)  NOT NULL WITH DEFAULT,
     LAST_REVIEW_DATE     DATE                                         ,
     REVIEW_PERCENT       UNSIGNED DECIMAL(6,3)                        ,
     PROMO_DATE           DATE                                         ,
     RETIRE_PLAN          CHAR(6)                                      ,
     RETIRE_PERCENT       UNSIGNED DECIMAL(6,3)                        ,
     BONUS_AMOUNT         UNSIGNED DECIMAL(10,2)                       ,
     COMP_ACCRUED         UNSIGNED DECIMAL(6,2)   NOT NULL WITH DEFAULT,
     COMP_TAKEN           UNSIGNED DECIMAL(6,2)   NOT NULL WITH DEFAULT,
     EDUC_LEVEL           CHAR(06)                                     ,
     UNION_ID             CHAR(10)                                     ,
     UNION_DUES           UNSIGNED DECIMAL(10,2)                       ,
 CHECK ( (RETIRE_PLAN IN ('STOCK', 'BONDS', '401K') ) AND
         (EDUC_LEVEL IN ('GED', 'HSDIP', 'JRCOLL', 'COLL',
          'MAS', 'PHD') ) ) )
 IN SQLDEMO.EMPLAREA;
```

```
CREATE TABLE          COVERAGE
    (PLAN_CODE         CHAR(03)                                NOT NULL,
     EMP_ID            UNSIGNED NUMERIC(4,0)                   NOT NULL,
     SELECTION_DATE    DATE                       NOT NULL WITH DEFAULT,
     TERMINATION_DATE  DATE                                            ,
     NUM_DEPENDENTS    UNSIGNED NUMERIC(2,0)      NOT NULL WITH DEFAULT)
 IN SQLDEMO.EMPLAREA;
CREATE TABLE          DEPARTMENT
    (DEPT_ID           UNSIGNED NUMERIC(4,0)                   NOT NULL,
     DEPT_HEAD_ID      UNSIGNED NUMERIC(4,0)                           ,
     DIV_CODE          CHAR(03)                                NOT NULL,
     DEPT_NAME         CHAR(40)                                NOT NULL)
IN SQLDEMO.INFOAREA;

CREATE TABLE          DIVISION
    (DIV_CODE          CHAR(03)                                NOT NULL,
     DIV_HEAD_ID       UNSIGNED NUMERIC(4,0)                           ,
     DIV_NAME          CHAR(40)                                NOT NULL)
 IN SQLDEMO.INFOAREA;

CREATE TABLE          EMPLOYEE
    (EMP_ID               UNSIGNED NUMERIC(4,0)                NOT NULL,
     MANAGER_ID           UNSIGNED NUMERIC(4,0)                        ,
     EMP_FNAME            CHAR(20)                             NOT NULL,
     EMP_LNAME            CHAR(20)                             NOT NULL,
     DEPT_ID              UNSIGNED NUMERIC(4,0)                NOT NULL,
     STREET               CHAR(40)                             NOT NULL,
     CITY                 CHAR(20)                             NOT NULL,
     STATE                CHAR(02)                             NOT NULL,
     ZIP_CODE             CHAR(09)                             NOT NULL,
     PHONE                CHAR(10)                                     ,
     STATUS               CHAR                                 NOT NULL,
     SS_NUMBER            UNSIGNED NUMERIC(9,0)                NOT NULL,
     START_DATE           DATE                                 NOT NULL,
     TERMINATION_DATE     DATE                                         ,
     BIRTH_DATE           DATE                                         ,
CHECK ( ( EMP_ID <= 8999 ) AND (STATUS IN ('A', 'S', 'L', 'T') ) ) )
IN SQLDEMO.EMPLAREA;
```

```
CREATE TABLE          INSURANCE_PLAN
     (PLAN_CODE            CHAR(03)                             NOT NULL,
      COMP_NAME            CHAR(40)                             NOT NULL,
      STREET               CHAR(40)                             NOT NULL,
      CITY                 CHAR(20)                             NOT NULL,
      STATE                CHAR(02)                             NOT NULL,
      ZIP_CODE             CHAR(09)                             NOT NULL,
      PHONE                CHAR(10)                             NOT NULL,
      GROUP_NUMBER         UNSIGNED NUMERIC(4,0)                NOT NULL,
      DEDUCT               UNSIGNED DECIMAL(9,2)                        ,
      MAX_LIFE_BENEFIT     UNSIGNED DECIMAL(9,2)                        ,
      FAMILY_COST          UNSIGNED DECIMAL(9,2)                        ,
      DEP_COST             UNSIGNED DECIMAL(9,2)                        ,
      EFF_DATE             DATE                                 NOT NULL)
  IN SQLDEMO.INFOAREA;
  CREATE TABLE          JOB
      (JOB_ID               UNSIGNED NUMERIC(4,0)               NOT NULL,
       JOB_TITLE            CHAR(20)                            NOT NULL,
       MIN_RATE             UNSIGNED DECIMAL(10,2)                      ,
       MAX_RATE             UNSIGNED DECIMAL(10,2)                      ,
       SALARY_IND           CHAR(01)                                    ,
       NUM_OF_POSITIONS     UNSIGNED DECIMAL(4,0)                       ,
       EFF_DATE             DATE                                        ,
       JOB_DESC_LINE_1      VARCHAR(60)                                 ,
       JOB_DESC_LINE_2      VARCHAR(60)                                 ,
   CHECK ( SALARY_IND IN ('S', 'H') ) )
   IN SQLDEMO.INFOAREA;

  CREATE TABLE          POSITION
      (EMP_ID               UNSIGNED NUMERIC(4,0)               NOT NULL,
       JOB_ID               UNSIGNED NUMERIC(4,0)               NOT NULL,
       START_DATE           DATE                                NOT NULL,
       FINISH_DATE          DATE                                        ,
       HOURLY_RATE        UNSIGNED   DECIMAL(7,2)                       ,
       SALARY_AMOUNT      UNSIGNED   DECIMAL(10,2)                      ,
       BONUS_PERCENT      UNSIGNED   DECIMAL(7,3)                       ,
       COMM_PERCENT       UNSIGNED   DECIMAL(7,3)                       ,
       OVERTIME_RATE      UNSIGNED   DECIMAL(5,2)                       ,
   CHECK ( (HOURLY_RATE IS NOT NULL AND SALARY_AMOUNT IS NULL)
           OR (HOURLY_RATE IS NULL AND SALARY_AMOUNT IS NOT NULL) ) )
   IN SQLDEMO.EMPLAREA;

   CREATE SCHEMA  DEMOPROJ;

   SET SESSION CURRENT SCHEMA DEMOPROJ;
```

```
 CREATE TABLE           ASSIGNMENT
     (EMP_ID             UNSIGNED NUMERIC(4,0)                  NOT NULL,
      PROJ_ID            CHAR(10)                               NOT NULL,
      START_DATE         DATE                                   NOT NULL,
      END_DATE           DATE                                           )
  IN PROJSEG.PROJAREA;
CREATE TABLE           CONSULTANT
    (CON_ID               UNSIGNED NUMERIC(4,0)                 NOT NULL,
     CON_FNAME            CHAR(20)                              NOT NULL,
     CON_LNAME            CHAR(20)                              NOT NULL,
     MANAGER_ID           UNSIGNED NUMERIC(4,0)                 NOT NULL,
     DEPT_ID              UNSIGNED NUMERIC(4,0)                 NOT NULL,
     PROJ_ID              CHAR(10)                                      ,
     STREET               CHAR(40)                                      ,
     CITY                 CHAR(20)                              NOT NULL,
     STATE                CHAR(02)                              NOT NULL,
     ZIP_CODE             CHAR(09)                              NOT NULL,
     PHONE                CHAR(10)                                      ,
     BIRTH_DATE           DATE                                          ,
     START_DATE           DATE                                  NOT NULL,
     SS_NUMBER            UNSIGNED NUMERIC(9,0)                 NOT NULL,
     RATE                 UNSIGNED DECIMAL(7,2)                         ,
  CHECK ( (CON_ID >= 9000 AND CON_ID <= 9999) ) )
  IN PROJSEG.PROJAREA;

 CREATE TABLE           EXPERTISE
     (EMP_ID               UNSIGNED NUMERIC(4,0)                NOT NULL,
      SKILL_ID             UNSIGNED NUMERIC(4,0)                NOT NULL,
      SKILL_LEVEL          CHAR(02)                                     ,
      EXP_DATE             DATE                                         )
  IN PROJSEG.PROJAREA;
```

```
CREATE TABLE            PROJECT
    (PROJ_ID               CHAR(10)                            NOT NULL,
     PROJ_LEADER_ID        UNSIGNED NUMERIC(4,0)                        ,
     EST_START_DATE        DATE                                         ,
     EST_END_DATE          DATE                                         ,
     ACT_START_DATE        DATE                                         ,
     ACT_END_DATE          DATE                                         ,
     EST_MAN_HOURS         UNSIGNED DECIMAL(7,2)                        ,
     ACT_MAN_HOURS         UNSIGNED DECIMAL(7,2)                        ,
     PROJ_DESC             VARCHAR(60)                         NOT NULL)
 IN PROJSEG.PROJAREA;

CREATE TABLE            SKILL
    (SKILL_ID              UNSIGNED NUMERIC(4,0)               NOT NULL,
     SKILL_NAME            CHAR(20)                            NOT NULL,
     SKILL_DESC            VARCHAR(60)                                  )
 IN PROJSEG.PROJAREA;


CREATE UNIQUE CALC KEY ON DEMOEMPL.DEPARTMENT(DEPT_ID);

CREATE UNIQUE CALC KEY ON DEMOEMPL.DIVISION(DIV_CODE);

CREATE UNIQUE CALC KEY ON DEMOEMPL.EMPLOYEE(EMP_ID);

CREATE UNIQUE CALC KEY ON DEMOEMPL.INSURANCE_PLAN(PLAN_CODE);

CREATE UNIQUE CALC KEY ON DEMOEMPL.JOB(JOB_ID);

CREATE UNIQUE CALC KEY ON DEMOPROJ.CONSULTANT(CON_ID);

CREATE UNIQUE CALC KEY ON DEMOPROJ.PROJECT(PROJ_ID);

CREATE UNIQUE CALC KEY ON DEMOPROJ.SKILL(SKILL_ID);
```

```
 CREATE UNIQUE INDEX AS_EMPROJ_NDX ON
        DEMOPROJ.ASSIGNMENT(EMP_ID,PROJ_ID);

 CREATE UNIQUE INDEX EX_EMPSKILL_NDX ON
        DEMOPROJ.EXPERTISE(EMP_ID, SKILL_ID);

 CREATE INDEX CO_CODE_NDX ON DEMOEMPL.COVERAGE(PLAN_CODE)
        IN SQLDEMO.INDXAREA;

 CREATE INDEX DE_CODE_NDX ON DEMOEMPL.DEPARTMENT(DIV_CODE);

 CREATE INDEX DI_HEAD_NDX ON DEMOEMPL.DIVISION(DIV_HEAD_ID);

 CREATE INDEX DE_HEAD_NDX ON DEMOEMPL.DEPARTMENT(DEPT_HEAD_ID);

 CREATE INDEX EM_MANAGER_NDX ON DEMOEMPL.EMPLOYEE(MANAGER_ID)
        IN SQLDEMO.INDXAREA;
 CREATE INDEX EM_NAME_NDX ON DEMOEMPL.EMPLOYEE(EMP_LNAME, EMP_FNAME)
        IN SQLDEMO.INDXAREA;

 CREATE INDEX EM_DEPT_NDX ON DEMOEMPL.EMPLOYEE(DEPT_ID)
        IN SQLDEMO.INDXAREA;

 CREATE INDEX IN_NAME_NDX ON DEMOEMPL.INSURANCE_PLAN(COMP_NAME)
         COMPRESSED;

 CREATE INDEX PO_JOB_NDX ON DEMOEMPL.POSITION(JOB_ID)
        IN SQLDEMO.INDXAREA;

CREATE INDEX CN_NAME_NDX ON DEMOPROJ.CONSULTANT(CON_LNAME,CON_FNAME);



  CREATE CONSTRAINT EMP_BENEFITS
      DEMOEMPL.BENEFITS  (EMP_ID)  REFERENCES
      DEMOEMPL.EMPLOYEE  (EMP_ID)
              LINKED CLUSTERED
              ORDER BY (FISCAL_YEAR DESC);
```

```
CREATE CONSTRAINT INSPLAN_COVERAGE
    DEMOEMPL.COVERAGE        (PLAN_CODE)  REFERENCES
    DEMOEMPL.INSURANCE_PLAN  (PLAN_CODE)
            UNLINKED;

CREATE CONSTRAINT EMP_COVERAGE
    DEMOEMPL.COVERAGE  (EMP_ID) REFERENCES
    DEMOEMPL.EMPLOYEE  (EMP_ID)
            LINKED CLUSTERED
            ORDER BY ( PLAN_CODE) UNIQUE;

CREATE CONSTRAINT DIVISION_DEPT
    DEMOEMPL.DEPARTMENT  (DIV_CODE)  REFERENCES
    DEMOEMPL.DIVISION    (DIV_CODE)
            UNLINKED;
CREATE CONSTRAINT EMP_DEPT_HEAD
    DEMOEMPL.DEPARTMENT  (DEPT_HEAD_ID)  REFERENCES
    DEMOEMPL.EMPLOYEE    (EMP_ID)
            UNLINKED;

CREATE CONSTRAINT EMP_DIV_HEAD
    DEMOEMPL.DIVISION  (DIV_HEAD_ID)  REFERENCES
    DEMOEMPL.EMPLOYEE  (EMP_ID)
            UNLINKED;

CREATE CONSTRAINT DEPT_EMPLOYEE
    DEMOEMPL.EMPLOYEE   (DEPT_ID)  REFERENCES
    DEMOEMPL.DEPARTMENT (DEPT_ID)
            UNLINKED;

CREATE CONSTRAINT MANAGER_EMP
    DEMOEMPL.EMPLOYEE  (MANAGER_ID)  REFERENCES
    DEMOEMPL.EMPLOYEE  (EMP_ID)
            UNLINKED;

CREATE CONSTRAINT SKILL_EXPERTISE
    DEMOPROJ.EXPERTISE  (SKILL_ID)  REFERENCES
    DEMOPROJ.SKILL      (SKILL_ID)
            LINKED CLUSTERED;
CREATE CONSTRAINT EMP_POSITION
    DEMOEMPL.POSITION  (EMP_ID)  REFERENCES
    DEMOEMPL.EMPLOYEE  (EMP_ID)
            LINKED CLUSTERED
            ORDER BY (JOB_ID) UNIQUE;
```

```
CREATE CONSTRAINT JOB_POSITION
    DEMOEMPL.POSITION  (JOB_ID)  REFERENCES
    DEMOEMPL.JOB       (JOB_ID)
            UNLINKED;

CREATE CONSTRAINT PROJECT_ASSIGN
    DEMOPROJ.ASSIGNMENT  (PROJ_ID)  REFERENCES
    DEMOPROJ.PROJECT   (PROJ_ID)
            LINKED CLUSTERED;

CREATE CONSTRAINT PROJECT_CONSULT
    DEMOPROJ.CONSULTANT  (PROJ_ID)  REFERENCES
    DEMOPROJ.PROJECT     (PROJ_ID)
            LINKED INDEX ORDER BY (PROJ_ID);


ALTER TABLE DEMOEMPL.COVERAGE
    DROP DEFAULT INDEX;

ALTER TABLE DEMOEMPL.DEPARTMENT
    DROP DEFAULT INDEX;

ALTER TABLE DEMOEMPL.DIVISION
    DROP DEFAULT INDEX;

ALTER TABLE DEMOEMPL.EMPLOYEE
    DROP DEFAULT INDEX;
ALTER TABLE DEMOEMPL.INSURANCE_PLAN
    DROP DEFAULT INDEX;

ALTER TABLE DEMOEMPL.POSITION
    DROP DEFAULT INDEX;

ALTER TABLE DEMOPROJ.ASSIGNMENT
    DROP DEFAULT INDEX;

ALTER TABLE DEMOPROJ.CONSULTANT
    DROP DEFAULT INDEX;

ALTER TABLE DEMOPROJ.EXPERTISE
    DROP DEFAULT INDEX;
```

```
CREATE VIEW DEMOEMPL.EMP_VACATION
       (EMP_ID, DEPT_ID, VAC_TIME)
       AS SELECT E.EMP_ID, DEPT_ID, SUM(VAC_ACCRUED) - SUM(VAC_TAKEN)
          FROM DEMOEMPL.EMPLOYEE E, DEMOEMPL.BENEFITS B
          WHERE E.EMP_ID = B.EMP_ID
          GROUP BY DEPT_ID, E.EMP_ID;

CREATE VIEW DEMOEMPL.OPEN_POSITIONS
       (JOB_ID, JOB_NAME, OPEN_POS)
       AS SELECT J.JOB_ID, J.JOB_TITLE,
                 (J.NUM_OF_POSITIONS - COUNT(P.JOB_ID))
          FROM DEMOEMPL.JOB J, DEMOEMPL.POSITION P
          WHERE P.FINISH_DATE IS NULL AND P.JOB_ID = J.JOB_ID
                PRESERVE DEMOEMPL.JOB
          GROUP BY J.JOB_ID, J.JOB_TITLE, J.NUM_OF_POSITIONS
          HAVING (J.NUM_OF_POSITIONS - COUNT(P.JOB_ID)) > 0;

CREATE VIEW DEMOEMPL.EMP_HOME_INFO
       AS SELECT EMP_ID, EMP_LNAME, EMP_FNAME, STREET, CITY, STATE,
                 ZIP_CODE, PHONE
          FROM DEMOEMPL.EMPLOYEE;

CREATE VIEW DEMOEMPL.EMP_WORK_INFO
       AS SELECT EMP_ID, MANAGER_ID, START_DATE, TERMINATION_DATE
          FROM DEMOEMPL.EMPLOYEE;
```

Appendix B: Sample Non-SQL Database Definition

Sample Database Schema Definition

```
ADD SCHEMA NAME IS EMPSCHM VERSION IS 100
SCHEMA DESCRIPTION IS 'EMPLOYEE DEMO DATABASE'
COMMENTS 'INSTALLATION: COMMONWEATHER CORPORATION'
    .
ADD AREA NAME IS EMP-DEMO-REGION
    .
ADD AREA NAME IS ORG-DEMO-REGION
    .
ADD AREA NAME IS INS-DEMO-REGION
    .
ADD RECORD NAME IS COVERAGE
    SHARE STRUCTURE OF RECORD COVERAGE VERSION IS 100
    RECORD ID IS 0400
    LOCATION MODE IS VIA            EMP-COVERAGE SET
    WITHIN AREA INS-DEMO-REGION
    OFFSET  5 PAGES FOR 45 PAGES
    .
ADD RECORD NAME IS DENTAL-CLAIM
    SHARE STRUCTURE OF RECORD DENTAL-CLAIM VERSION IS 100
    RECORD ID IS 0405
    LOCATION MODE IS VIA         COVERAGE-CLAIMS SET
    WITHIN AREA INS-DEMO-REGION
    OFFSET  5 PAGES FOR 45 PAGES
    MINIMUM ROOT LENGTH IS            130 CHARACTERS
    MINIMUM FRAGMENT LENGTH IS         RECORD LENGTH
    .
```

```
ADD RECORD NAME IS DEPARTMENT
    SHARE STRUCTURE OF RECORD DEPARTMENT VERSION IS 100
    RECORD ID IS 0410
    LOCATION MODE IS CALC          USING DEPT-ID-0410
                               DUPLICATES NOT ALLOWED
    WITHIN AREA ORG-DEMO-REGION
    OFFSET  5 PAGES FOR 45 PAGES
    .
ADD RECORD NAME IS EMPLOYEE
    SHARE STRUCTURE OF RECORD EMPLOYEE VERSION IS 100
    RECORD ID IS 0415
    LOCATION MODE IS CALC           USING EMP-ID-0415
                               DUPLICATES NOT ALLOWED
    WITHIN AREA EMP-DEMO-REGION
    OFFSET  5 PAGES FOR 95 PAGES
    .
ADD RECORD NAME IS EMPOSITION
    SHARE STRUCTURE OF RECORD EMPOSITION VERSION IS 100
    RECORD ID IS 0420
    LOCATION MODE IS VIA           EMP-EMPOSITION SET
    WITHIN AREA EMP-DEMO-REGION
    OFFSET  5 PAGES FOR 95 PAGES
  .
ADD RECORD NAME IS EXPERTISE
    SHARE STRUCTURE OF RECORD EXPERTISE VERSION IS 100
    RECORD ID IS 0425
    LOCATION MODE IS VIA            EMP-EXPERTISE SET
    WITHIN AREA EMP-DEMO-REGION
    OFFSET  5 PAGES FOR 95 PAGES
    .
ADD RECORD NAME IS HOSPITAL-CLAIM
    SHARE STRUCTURE OF RECORD HOSPITAL-CLAIM VERSION IS 100
    RECORD ID IS 0430
    LOCATION MODE IS VIA          COVERAGE-CLAIMS SET
    WITHIN AREA INS-DEMO-REGION
    OFFSET  5 PAGES FOR 45 PAGES
    .
ADD RECORD NAME IS INSURANCE-PLAN
    SHARE STRUCTURE OF RECORD INSURANCE-PLAN VERSION IS 100
    RECORD ID IS 0435
    LOCATION MODE IS CALC    USING INS-PLAN-CODE-0435
                               DUPLICATES NOT ALLOWED
    WITHIN AREA INS-DEMO-REGION
    OFFSET  1 PAGE  FOR  4 PAGES
    .
```

```
ADD RECORD NAME IS JOB
    SHARE STRUCTURE OF RECORD JOB VERSION IS 100
    RECORD ID IS 0440
    LOCATION MODE IS CALC           USING JOB-ID-0440
                               DUPLICATES NOT ALLOWED
    WITHIN AREA ORG-DEMO-REGION
    OFFSET  5 PAGES FOR 45 PAGES
    MINIMUM ROOT LENGTH IS CONTROL LENGTH
    MINIMUM FRAGMENT LENGTH IS RECORD LENGTH
    CALL IDMSCOMP BEFORE STORE
    CALL IDMSCOMP BEFORE MODIFY
    CALL IDMSDCOM AFTER GET
   .
ADD RECORD NAME IS NON-HOSP-CLAIM
    SHARE STRUCTURE OF RECORD NON-HOSP-CLAIM VERSION IS 100
    RECORD ID IS 0445
    LOCATION MODE IS VIA          COVERAGE-CLAIMS SET
    WITHIN AREA INS-DEMO-REGION
    OFFSET  5 PAGES FOR 45 PAGES
    MINIMUM ROOT LENGTH IS             248 CHARACTERS
    MINIMUM FRAGMENT LENGTH IS          RECORD LENGTH
    .
ADD RECORD NAME IS OFFICE
    SHARE STRUCTURE OF RECORD OFFICE VERSION IS 100
    RECORD ID IS 0450
    LOCATION MODE IS CALC      USING OFFICE-CODE-0450
                               DUPLICATES NOT ALLOWED
    WITHIN AREA ORG-DEMO-REGION
    OFFSET  5 PAGES FOR 45 PAGES
    .
ADD RECORD NAME IS SKILL
    SHARE STRUCTURE OF RECORD SKILL VERSION IS 100
    RECORD ID IS 0455
    LOCATION MODE IS CALC         USING SKILL-ID-0455
                               DUPLICATES NOT ALLOWED
    WITHIN AREA ORG-DEMO-REGION
    OFFSET  5 PAGES FOR 45 PAGES
    .
```

```
ADD RECORD NAME IS STRUCTURE
    SHARE STRUCTURE OF RECORD STRUCTURE VERSION IS 100
    RECORD ID IS 0460
    LOCATION MODE IS VIA                  MANAGES SET
    WITHIN AREA EMP-DEMO-REGION
    OFFSET  5 PAGES FOR 95 PAGES
   .
ADD SET NAME IS COVERAGE-CLAIMS
    ORDER IS LAST
    MODE IS CHAIN LINKED TO PRIOR
    OWNER IS COVERAGE
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
    MEMBER IS HOSPITAL-CLAIM
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
        MANDATORY AUTOMATIC
    MEMBER IS NON-HOSP-CLAIM
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
        MANDATORY AUTOMATIC
    MEMBER IS DENTAL-CLAIM
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
        MANDATORY AUTOMATIC
    .
ADD SET NAME IS DEPT-EMPLOYEE
    ORDER IS SORTED
    MODE IS CHAIN LINKED TO PRIOR
    OWNER IS DEPARTMENT
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
    MEMBER IS EMPLOYEE
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
        LINKED TO OWNER
            OWNER DBKEY POSITION IS AUTO
        OPTIONAL AUTOMATIC
        ASCENDING KEY IS ( EMP-LAST-NAME-0415
                           EMP-FIRST-NAME-0415 )
            DUPLICATES LAST
   .
```

```
ADD SET NAME IS EMP-COVERAGE
    ORDER IS FIRST
    MODE IS CHAIN LINKED TO PRIOR
    OWNER IS EMPLOYEE
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
    MEMBER IS COVERAGE
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
        LINKED TO OWNER
            OWNER DBKEY POSITION IS AUTO
        MANDATORY AUTOMATIC
    .
ADD SET NAME IS EMP-EMPOSITION
    ORDER IS FIRST
    MODE IS CHAIN LINKED TO PRIOR
    OWNER IS EMPLOYEE
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
    MEMBER IS EMPOSITION
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
        LINKED TO OWNER
            OWNER DBKEY POSITION IS AUTO
        MANDATORY AUTOMATIC
    .
ADD SET NAME IS EMP-EXPERTISE
    ORDER IS SORTED
    MODE IS CHAIN LINKED TO PRIOR
    OWNER IS EMPLOYEE
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
    MEMBER IS EXPERTISE
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
        LINKED TO OWNER
            OWNER DBKEY POSITION IS AUTO
        MANDATORY AUTOMATIC
        DESCENDING KEY IS (SKILL-LEVEL-0425 )
            DUPLICATES FIRST
  .
```

```
ADD SET NAME IS EMP-NAME-NDX
    ORDER IS SORTED
    MODE IS INDEX BLOCK CONTAINS 40 KEYS
    OWNER IS SYSTEM
        WITHIN AREA EMP-DEMO-REGION
        OFFSET  1 PAGE  FOR  4 PAGES
    MEMBER IS EMPLOYEE
        INDEX DBKEY POSITION IS AUTO
        OPTIONAL AUTOMATIC
        ASCENDING KEY IS ( EMP-LAST-NAME-0415
                           EMP-FIRST-NAME-0415 )
            COMPRESSED
            DUPLICATES LAST
    .
ADD SET NAME IS JOB-EMPOSITION
    ORDER IS NEXT
    MODE IS CHAIN LINKED TO PRIOR
    OWNER IS JOB
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
    MEMBER IS EMPOSITION
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
        LINKED TO OWNER
            OWNER DBKEY POSITION IS AUTO
        OPTIONAL MANUAL
    .
```

```
ADD SET NAME IS JOB-TITLE-NDX
    ORDER IS SORTED
    MODE IS INDEX BLOCK CONTAINS 30 KEYS
    OWNER IS SYSTEM
        WITHIN AREA ORG-DEMO-REGION
        OFFSET  1 PAGE  FOR  4 PAGES
    MEMBER IS JOB
        INDEX DBKEY POSITION IS AUTO
        OPTIONAL AUTOMATIC
        ASCENDING KEY IS ( TITLE-0440 )
            DUPLICATES NOT ALLOWED
 .
ADD SET NAME IS MANAGES
    ORDER IS NEXT
    MODE IS CHAIN LINKED TO PRIOR
    OWNER IS EMPLOYEE
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
    MEMBER IS STRUCTURE
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
        LINKED TO OWNER
            OWNER DBKEY POSITION IS AUTO
        MANDATORY AUTOMATIC
    .
ADD SET NAME IS OFFICE-EMPLOYEE
    ORDER IS SORTED
    MODE IS INDEX BLOCK CONTAINS 30 KEYS
    OWNER IS OFFICE
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
    MEMBER IS EMPLOYEE
        INDEX DBKEY POSITION IS AUTO
        LINKED TO OWNER
            OWNER DBKEY POSITION IS AUTO
        OPTIONAL AUTOMATIC
        ASCENDING KEY IS ( EMP-LAST-NAME-0415
                           EMP-FIRST-NAME-0415 )
            COMPRESSED
            DUPLICATES LAST
    .
```

```
ADD SET NAME IS REPORTS-TO
    ORDER IS NEXT
    MODE IS CHAIN LINKED TO PRIOR
    OWNER IS EMPLOYEE
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
    MEMBER IS STRUCTURE
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
        LINKED TO OWNER
            OWNER DBKEY POSITION IS AUTO
        OPTIONAL MANUAL
  .
ADD SET NAME IS SKILL-EXPERTISE
    ORDER IS SORTED
    MODE IS INDEX BLOCK CONTAINS 30 KEYS
    OWNER IS SKILL
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
    MEMBER IS EXPERTISE
        INDEX DBKEY POSITION IS AUTO
        LINKED TO OWNER
            OWNER DBKEY POSITION IS AUTO
        MANDATORY AUTOMATIC
        DESCENDING KEY IS ( SKILL-LEVEL-0425 )
            DUPLICATES FIRST
    .
ADD SET NAME IS SKILL-NAME-NDX
    ORDER IS SORTED
    MODE IS INDEX BLOCK CONTAINS 30 KEYS
    OWNER IS SYSTEM
        WITHIN AREA ORG-DEMO-REGION
        OFFSET  1 PAGE  FOR  4 PAGES
    MEMBER IS SKILL
        INDEX DBKEY POSITION IS AUTO
        OPTIONAL AUTOMATIC
        ASCENDING KEY IS ( SKILL-NAME-0455 )
            DUPLICATES NOT ALLOWED
    .
VALIDATE
    .
```

Sample Database Subschema Definition

```
ADD SUBSCHEMA NAME IS EMPSS01
    OF SCHEMA NAME IS EMPSCHM VERSION  100
COMMENTS 'THIS IS THE COMPLETE VIEW OF EMPSCHM'
    .
ADD AREA NAME IS EMP-DEMO-REGION
    .
ADD AREA NAME IS INS-DEMO-REGION
    .
ADD AREA NAME IS ORG-DEMO-REGION
    .
ADD RECORD NAME IS COVERAGE
    .
ADD RECORD NAME IS DENTAL-CLAIM
    .
ADD RECORD NAME IS DEPARTMENT
    .
ADD RECORD NAME IS EMPLOYEE
    .
ADD RECORD NAME IS EMPOSITION
    .
ADD RECORD NAME IS EXPERTISE
    .
ADD RECORD NAME IS HOSPITAL-CLAIM
    .
ADD RECORD NAME IS INSURANCE-PLAN
    .
ADD RECORD NAME IS JOB
    .
ADD RECORD NAME IS NON-HOSP-CLAIM
    .
ADD RECORD NAME IS OFFICE
    .
ADD RECORD NAME IS SKILL
    .
ADD RECORD NAME IS STRUCTURE
   .
ADD SET COVERAGE-CLAIMS
    .
ADD SET DEPT-EMPLOYEE
    .
ADD SET EMP-COVERAGE
    .
ADD SET EMP-EXPERTISE
    .
ADD SET EMP-NAME-NDX
    .
```

```
ADD SET EMP-EMPOSITION
    .
ADD SET JOB-EMPOSITION
    .
ADD SET JOB-TITLE-NDX
    .
ADD SET MANAGES
    .
ADD SET OFFICE-EMPLOYEE
    .
ADD SET REPORTS-TO
    .
ADD SET SKILL-EXPERTISE
    .
ADD SET SKILL-NAME-NDX
    .
GENERATE
    .
```

Appendix C: Native VSAM Considerations

This section contains the following topics:

Overview

An overview of batch compilation and the JCL/commands you need to execute non-SQL schema and subschema statements under the central version or in local mode are discussed in this section.

Native VSAM Data Set Structures

You can structure a native VSAM data set as a key-sequenced data set (KSDS), an entry-sequenced data set (ESDS), or a relative record data set (RRDS). The following table lists the characteristics of each data set structure.

Native VSAM Data Set Structures

VSAM Structure 1	Access method	Record Format	
KSDS	■ Prime index ■ Alternate indexes	Fixed or variable	Spanned or nonspanned
ESDS	■ Relative byte address (RBA) ■ Alternate indexes	Fixed or variable	Spanned or nonspanned
RRDS	Relative record number	Fixed	Nonspanned

1 KSDS = key-sequenced data set

ESDS = entry-sequenced data set

RRDS = relative record data set

CA IDMS/DB Native VSAM Definitions

To use native VSAM files in a CA IDMS environment, you define native VSAM structures in the schema, segment, and DMCL. Schema, segment, and DMCL definitions are discussed separately as follows.

Schema Definition

The schema establishes the correspondences between the logical characteristics of the native VSAM file and CA IDMS/DB, as follows:

A File

A CA IDMS *file* represents a VSAM cluster or path:

- A KSDS with a prime index or alternate indexes or both
- An ESDS with or without alternate indexes
- An RRDS

An *area* represents a KSDS, ESDS, or RRDS data component. You map one area to each VSAM file; each area must have a unique page range. The page range is a function of the VSAM data set structure. For more information about how to determine the page range, see "AREA statements" in Physical Database DDL Statements.

A Schema Record

A schema *record* represents a VSAM data record:

- All VSAM data records can be represented as a record with a location mode of VSAM
- A KSDS with a prime index or alternate indexes and an ESDS with an alternate index can be represented as a CALC record

A Schema Set

A schema *set* represents the index of a VSAM data set and related records. These sets are sorted and maintained through the following types of native VSAM data set structures:

- A KSDS with a prime or alternate index
- An ESDS with an alternate index

You define sets with alternate indexes in the schema to allow record occurrences with duplicate sort keys. These record occurrences are retrieved in the order in which the records were stored, regardless of the order in which the set is searched. For sorted sets that do not allow duplicate sort keys, you can use any index to maintain the set.

Native VSAM sets allow you to code application programs that:

- Access a specific record directly by sort key
- Access records by generic sort key
- Process the native VSAM file in sort-key sequence from the start of the file or from a specified starting point.

Relationships Between VSAM and CA IDMS/DB Structures

The schema, AREA, RECORD, SET, and SET statements, and the segment statements needed to represent native VSAM structures are listed in the following table.

VSAM Structure	CA IDMS/DB Structure	DDL Statement
KSDS ESDS RRDS PATH	File	CREATE FILE *segment.file-name*
Data component: KSDS, ESDS, RRDS	Area	Schema definition: ADD AREA NAME IS *area-name* ... Segment definition: CREATE AREA *segment.area-name* ...
VSAM data record	Record	ADD RECORD NAME IS *record-name* LOCATION MODE IS VSAM VSAM TYPE IS *type* WITHIN AREA *area-name*.
VSAM data record: KSDS with prime or alternate index ESDS alternate index	CALC record	ADD RECORD NAME IS *record-name* LOCATION MODE IS VSAM CALC USING *calc-element-name* DUPLICATES ARE *duplicates-option* VSAM TYPE IS *type* WITHIN AREA *area-name*.

VSAM Structure	CA IDMS/DB Structure	DDL Statement
KSDS prime or alternate index ESDS alternate index	Set (sorted by prime or alternate key)	ADD SET NAME IS *set-name* MODE IS VSAM INCLUDE MEMBER IS *record-name* MANDATORY AUTOMATIC ASCENDING KEY IS *sort-key-name*.

DMCL Definition

The DMCL module establishes the correspondences at runtime between physical characteristics of the database and the native VSAM files. You describe native VSAM files to be accessed by CA IDMS/DB in the DMCL BUFFER statements with the following:

- PAGE CONTAINS specifies the size of the largest control interval of any native VSAM file associated with the buffer
- BUFFER CONTAINS specifies the number of I/O buffers in the buffer pool to be used to transfer records between memory and auxiliary storage
- NATIVE VSAM specifies that the buffer pool is for use exclusively with native VSAM files

DML Functions with Native VSAM

To access information from a native VSAM data set, CA IDMS/DB converts DML statements issued by the application program into record-level (not control-interval) VSAM macro variations (for example, ACB, RPL) and passes control to VSAM. No changes have to be made to the VSAM data set. A local run unit or central version appears to VSAM as a single application that opens VSAM data clusters, activates VSAM paths using local-shared resources (LSR) or nonshared resources (NSR), accesses data records, and closes the clusters and paths.

The following table lists different VSAM structures and the CA IDMS/DB DML functions that can be used to access the VSAM structures.

DML Functions for Native VSAM Data Set Access

CA IDMS/DB DML Statement	VSAM Structure
STORE last within area	ESDS
STORE direct by db-key	RRDS
STORE physical sequential	KSDS

CA IDMS/DB DML Statement	VSAM Structure
ERASE	KSDS or RRDS
FIND/OBTAIN FIRST/NEXT WITHIN SET	KSDS or ESDS with a primary index or alternate indexes
FIND/OBTAIN LAST/PRIOR WITHIN SET	KSDS or ESDS with a primary index or alternate indexes
FIND/OBTAIN WITHIN SET USING SORT KEY	KSDS or ESDS
FIND/OBTAIN FIRST/NEXT WITHIN AREA	KSDS, ESDS, or RRDS
FIND/OBTAIN LAST/PRIOR WITHIN AREA	KSDS, ESDS, or RRDS
FIND/OBTAIN CALC	KSDS or ESDS with a primary index or alternate indexes
FIND/OBTAIN CALC DUPLICATE	KSDS or ESDS with a primary index or alternate indexes
FIND/OBTAIN DB-KEY	ESDS or RRDS
MODIFY, changing CALC key or sort key	KSDS or ESDS with a primary index or alternate indexes
MODIFY, without changing CALC key or sort key	KSDS, ESDS, or RRDS
MODIFY, changing record length	KSDS
ROLLBACK following STORE (without restore and rollforward)	KSDS or RRDS
ROLLBACK following ERASE (without restore and rollforward)	KSDS or RRDS
ROLLBACK following MODIFY (without restore and rollforward)	KSDS or RRDS

Appendix D: Batch Compiler Execution JCL

This section contains the following topics:

Overview

CA IDMS/DB can access information from native VSAM data sets. A native VSAM data set is one that is defined to VSAM and contains VSAM records. Although a native VSAM data set is not structured as a CA IDMS database, it can be accessed as if it were.

Note: Native VSAM files are different from *database* files that have VSAM as an access method.

This appendix describes:

- Native VSAM data set structures
- Schema, segment, and DMCL considerations
- DML functions that can be used to access native VSAM structures

Batch Compilation

Local Mode Considerations

The DDL compilers can be run under the central version or in local mode. If the central version has access to the DDLDML or DDLDCLOD area of the dictionary in update usage mode, an attempt to execute a compiler in local mode without specifying USAGE RETRIEVAL in a SIGNON statement will terminate with an error code of 0966.

To ensure the integrity of the dictionary, either journal local mode compilations or back up the dictionary *before each local mode compilation*. The central version uses automatic recovery procedures to ensure the integrity of the dictionary.

Naming the Dictionary and System

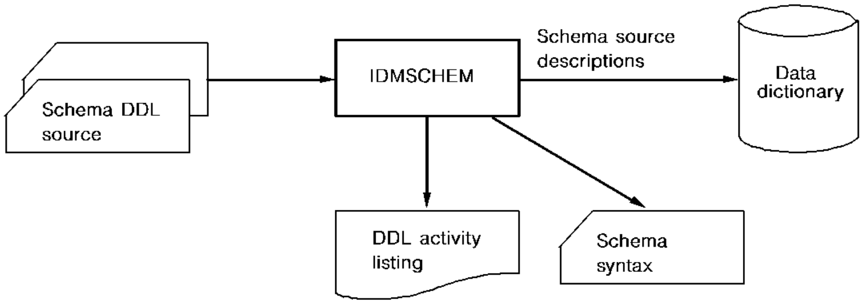
In an operating environment with multiple dictionaries, the name of the dictionary to be accessed (or the DC/UCF system on which it resides) can be specified through SYSIDMS parameters or on the compiler SIGNON statement or the command facility CONNECT statement. If specified in both places, the SIGNON or CONNECT specification takes precedence. If specified in the SYSIDMS file, the name of the dictionary is specified using DICTNAME and the DC/UCF system on which it resides is specified using DICTNODE.

Note: For more information about the SYSIDMS parameter file, see Chapter 25, “Dictionaries and Runtime Environments".

Compiling a Non-SQL Defined Schema

To compile a schema in batch mode, execute the program IDMSCHEM. Input and output are as follows:

Mode	Description
Input	Schema source statements
Output	■ A source description of the schema stored in the dictionary ■ A Schema Compiler Activity List A card image file containing schema syntax, if the source input contains a PUNCH statement

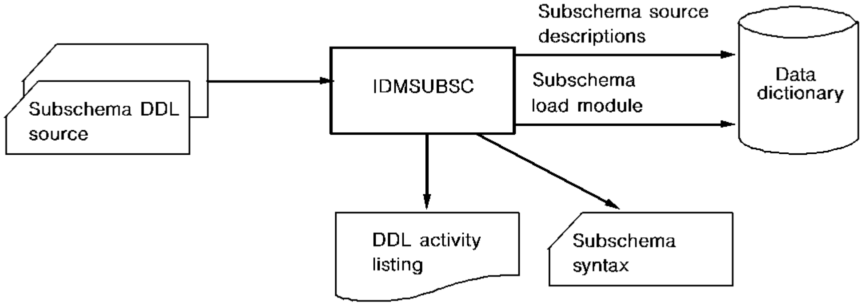


Note: To compile an SQL-defined schema, you submit SQL DDL statements through the CA IDMS Command Facility. For sample job streams, see the *CA IDMS Common Facilities Guide*.

Compiling a Subschema

To compile a subschema in batch mode, execute the program IDMSUBSC. Input and output are as follows:

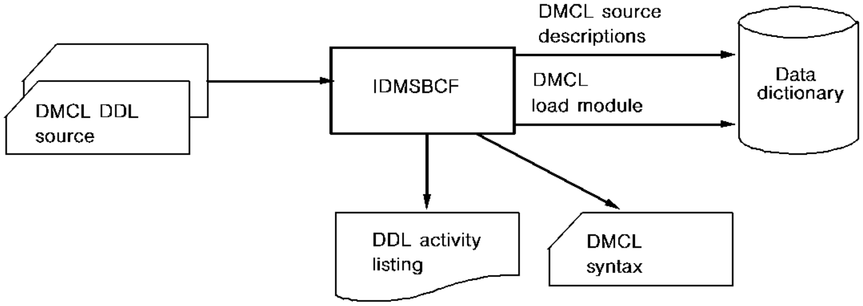
Mode	Description
Input	Subschema source statements
Output	■ A source description of the subschema stored in the dictionary ■ A Subschema Compiler Activity List ■ A subschema load module stored in the dictionary load area (DDLDCLOD), if the source input contains a GENERATE statement A card image file containing schema syntax, if the source input contains a PUNCH statement



Defining Segments, DMCLs, and Database Name Tables

To define a DMCL in batch mode, execute the program IDMSBCF (the CA IDMS Command Facility). Input and output are as follows:

Mode	Description
Input	Segments, DMCL, and database name table source statements as described in Physical Database DDL Statements.
Output	■ Segment, DMCL, and database name table sourcedescriptions stored in the dictionary ■ A DMCL or database name table load module, if the source contains a GENERATE statement ■ A command facility activity listing ■ A card image file containing DDL syntax or DMCL or database name table object code, if the source input contains a punch statement



Note: For more information about sample IDMSBCF job streams, see the *CA IDMS Common Facilities Guide*.

z/OS JCL

This section provides the z/OS JCL you need to run the schema and subschema compilers (central version and local mode).

Schema Compiler

IDMSCHEM—Central Version IDMSCHEM (z/OS)

```
//SCHEMA   EXEC PGM=IDMSCHEM,REGION=1024K
//STEPLIB   DD  DSN=idms.dba.loadlib,DISP=SHR
//          DD  DSN=idms.CUSTOM.LOADLIB,DISP=SHR
//          DD  DSN=idms.CAGJLOAD,DISP=SHR
//sysctl    DD  DSN=idms.sysctl,DISP=SHR
//dcmsg    DD  DSN=idms.sysmsg.ddldcmsg,DISP=SHR
//SYSLST    DD  SYSOUT=A
//SYSPCH    DD  DSN=&.&PCH., DISP=(NEW,PASS,DELETE),
                DCB=(RECFM=FB,BLKSIZE=9040,LRECL=80),
                     SPACE=space-specification,
                     UNIT=unit
//SYSIDMS   DD  *
Input SYSIDMS parameters, as required
/*
//SYSIPT    DD  *
Schema DDL source statements
```

Note: The SYSPCH DD statement is required only if the DDL specifies PUNCH TO SYSPCH.

idms.dba.loadlib

Data set name of the load library containing the DMCL and database name table load modules

idms.CUSTOM.LOADLIB

Data set name of the load library containing the customized CA IDMS executable modules

idms.CAGJLOAD

Data set name of the load library containing the vanilla CA IDMS executable modules

sysctl

DDname of the SYSCTL file

idms.sysctl

Data set name of the SYSCTL file

dcmsg

DDname of the message (DDLDCMSG) area

idms.sysmsg.ddldcmsg

Data set name of the message (DDLDCMSG) area

space-specification

Space specification for the punch file

unit

Symbolic device name

IDMSCHEM—Local Lode

To execute the schema compiler in local mode, remove the SYSCTL DD statement and replace it with:

```
//dictdb     DD  DSN=idms.appldict.ddldml,DISP=SHR
//dloddb     DD  DSN=idms.appldict.ddldclod,DISP=SHR
//sysjrnl    DD  DSN=idms.tapejrnl,DISP=(NEW,KEEP),
//               UNIT=tape
Additional journal file assignments, as required
```

Note: Include the dloddb DD statement only if the DDL contains the REGENERATE statement.

dictdb

Ddname of the data dictionary DDLDML area

dloddb

Ddname of the data dictionary load area

idms.appldict.ddldclod

Data set name of the tape journal file

idms.appldict.ddldml

Data set name of the data dictionary DDLDML area

sysjrnl

Ddname of the tape journal file; must be appropriate for the DMCL module being used

tape

Symbolic device name for the tape journal file

Subschema Compiler

IDMSUBSC—Central Version IDMSUBSC (z/OS)

```
//SUBSCHEM EXEC PGM=IDMSUBSC,REGION=1024K
//STEPLIB   DD  DSN=idms.dba.loadlib,DISP=SHR
//          DD  DSN=idms.CUSTOM.LOADLIB,DISP=SHR
//          DD  DSN=idms.CAGJLOAD
//sysctl    DD  DSN=idms.sysctl,DISP=SHR
//dcmsg    DD  DSN=idms.sysmsg.ddldcmsg,DISP=SHR
//SYSLST    DD  SYSOUT=A
//SYSPCH    DD  DSN=&.&PCH., DISP=(NEW,KEEP,DELETE),
                DCB=(RECFM=FB,BLKSIZE=9040,LRECL=80),
                     SPACE=space-specification,
                     UNIT=unit
//SYSIDMS   DD  *
Input SYSIDMS parameters, as required
/*
//SYSIPT    DD  *
Subschema DDL source statements
```

Note: The SYSPCH DD statement is required only if the DDL specifies PUNCH TO SYSPCH.

idms.dba.loadlib

Data set name of the load library containing the DMCL and database name table load modules

idms.CUSTOM.LOADLIB

Data set name of the load library containing the customized CA IDMS executable modules

idms.CAGJLOAD

Data set name of the load library containing the vanilla CA IDMS executable modules

sysctl

DDname of the SYSCTL file

idms.sysctl

Data set name of the SYSCTL file

dcmsg

DDname of the message (DDLDCMSG) area

idms.sysmsg.ddldcmsg

Data set name of the message (DDLDCMSG) area

space-specification unit

See IDMSCHEM job stream

IDMSUBSC—Local Mode

To execute the subschema compiler in local mode, remove the SYSCTL DD statement and replace it with:

```
//dictdb     DD  DSN=idms.appldict.ddldml,DISP=SHR
//dloddb     DD  DSN=idms.appldict.ddldclod,DISP=SHR
//sysjrnl    DD  DSN=idms.tapejrnl,DISP=(NEW,KEEP),
//               UNIT=tape
Additional journal file assignments, as required
```

Note: Include the dloddb DD statement only if the DDL contains the REGENERATE statement.

dictdb

DDname of the dictionary DDLDML area

idms.appldict.ddldml

Data set name of the dictionary DDLDML area

dloddb

DDname of the dictionary load area

idms.appldict.ddldclod

Data set name of the dictionary load area

sysjrnl

DDname of the tape journal file; must be appropriate for the DMCL module being used

idms.tapejrnl

Data set of the tape journal file

tape

Symbolic device name

z/VSE JCL

The following z/VSE information is presented in this section:

- =COPY facility for input parameter statements
- z/VSE JCL to run the schema and subschema compilers (central version and local mode)

=COPY Facility

Purpose

Under z/VSE, some or all of the input parameter statement to be submitted to a DDL compiler can be stored as a member in a source statement library. To copy the library member into the job stream, you use the =COPY IDMS statement.

The =COPY IDMS statement identifies the library member and is coded in the JCL along with other input parameter statements (if any) to be submitted to the DDL compiler. Multiple =COPY statements can be submitted.

=COPY IDMS statements and input parameter statements can be intermixed in the JCL. The input parameters are submitted to the compiler in the order in which they occur, whether they are coded directly in the JCL or copied in through the =COPY facility.

Syntax

```
►►─── =COPY IDMS ─┬──────────────────┬─ member-name ──────────────────────►◄
                  ├─ A. ◄ ───────────┤
                  └─ sublibrary-id. ─┘
```

Parameters

A/sublibrary-id

Identifies the source statement sublibrary that includes the member identified by *member-name*. The default is A.

member-name

Identifies the source statement library member that contains the input parameter statements to be submitted to the compiler.

Note: If the input parameter statements are stored as a member in a private source statement library, the DLBL file type for the library must be specified as DA.

Schema Compiler

IDMSCHEM—Central Version IDMSCHEM (z/VSE)

```
// EXEC PROC=IDMSLBLS
// UPSI     b   If specified in the IDMSOPTI module
// DLBL     idmspch,'temp.ddl',0
// EXTENT   sys020,nnnnnn,,,ssss,llll
   ASSGN    sys020,DISK,VOL=nnnnnn,SHR
// EXEC     IDMSCHEM
Optional SYSIDMS parameters
/*
Schema DDL source statements
/*
```

Include the DLBL, EXTENT, and ASSGN statements for IDMSPCH only if the DDL specifies PUNCH TO SYSPCH. See the *CA IDMS System Operations Guide* for details.

Overriding IDMSOPTI

At installation, you can define a SYSCTL procedure that overrides the IDMSOPTI specifications for central version operations.

Note: For more information about the SYSCTL procedure, see the *CA IDMS Installation and Maintenance Guide—z/VSE*.

IDMSLBLS

Name of the procedure provided at installation that contains the file definitions for CA IDMS dictionaries and databases.

Note: For a complete listing of IDMSLBLS, see E.4.6, “IDMSLBLS Procedure”.

IDMSLBLS references SYSIDMS, the input file you can use to specify runtime parameters, such as DMCL or dictionary name.

Note: For more information about SYSIDMS parameters, see the CA IDMS Common Facilities Guide or CA IDMS Navigational DML Programming Guide.

b

Appropriate UPSI switch, 1-8 characters, as specified in the IDMSOPTI module

idmspch

Filename of the punched output (from IDMSPCH)

temp.ddl

File ID of the punched output (from IDMSPCH)

nnnnnn

Serial number of the disk volume

ssss

Starting track (CKD) or block (FBA) of disk extent

llll

Number of the tracks (CKD) or blocks (FBA) of disk extent

sys020

Logical unit assignment of the punched output

IDMSCHEM—Local Mode

To execute the schema compiler in local mode, remove the UPSI specification, and include the following statements before EXEC IDMSCHEM:

```
// TLBL     sysjrnl,'idms.tapejrnl',,nnnnnn,,f
// ASSGN    sys009,TAPE,VOL=nnnnnn
```

sysjrnl

File name of the tape journal file

idms.tapejrnl

File ID of the tape journal file

nnnnnn

Volume serial number

f

File number of the tape journal file

sys009

Logical unit assignment for the tape journal file

Subschema Compiler

IDMSUBSC—Central Version IDMSUBSC (z/VSE)

```
// EXEC PROC=IDMSLBLS
// UPSI     b      If specified in the IDMSOPTI module
// DLBL     idmspch,'temp.ddl',0
// EXTENT   sys020,nnnnnn,,,ssss,llll
   ASSGN    sys020,DISK,VOL=nnnnnn,SHR
// EXEC     IDMSUBSC
Optional SYSIDMS parameters
/*
Subschema DDL source statements
/*
```

Include the DLBL, EXTENT, and ASSGN statements for IDMSPCH only if the DDL specifies PUNCH TO SYSPCH. To route punched output to a sequential disk file or to a tape file, use SYSIDMS file parameters to override the default characteristics, if necessary. See the *CA IDMS System Operations Guide* for details.

Overriding IDMSOPTI

At installation, you can define a SYSCTL procedure that overrides the IDMSOPTI specifications for central version operations.

Note: For more information about the SYSCTL procedure, see the *CA IDMS Installation and Maintenance Guide—z/VSE*.

IDMSLBLS

Name of the procedure provided at installation that contains the file definitions for CA IDMS dictionaries and databases.

Note: For a complete listing of IDMSLBLS, see IDMSLBLS Procedure.

IDMSLBLS references SYSIDMS, the input file you can use to specify runtime parameters, such as DMCL or dictionary name.

Note: For more information about SYSIDMS parameters, see the CA IDMS Common Facilities Guide or CA IDMS Navigational DML Programming Guide.

b

Appropriate UPSI switch, 1-8 characters, as specified in the IDMSOPTI module

idmspch

Filename of the punched output (from IDMSPCH)

temp.ddl

File ID of the punched output (from IDMSPCH)

sys020

Logical unit assignment of the punched output disk extent

nnnnnn

Volume serial number

ssss

Starting track (CKD) or block (FBA) of the disk extent

llll

Number of the tracks (CKD) or blocks (FBA) of the disk extent

sysctl

Filename of the SYSCTL file

idms.sysctl

File ID of the SYSCTL file

sys008

Logical unit assignment of the SYSCTL file

IDMSUBSC—Local Mode

To execute the subschema compiler in local mode, remove the UPSI specification, and include the following statements before EXEC IDMSUBSC:

```
// TLBL     sysjrnl,'idms.tapejrnl',,nnnnnn,,f
// ASSGN    sys009,TAPE,VOL=nnnnnn
```

Note: These variables are described under the local mode discussion for the IDMSCHEM job stream.

IDMSLBLS Procedure

IDMSLBLS is a procedure that contains file definitions for the dictionaries, sample databases, disk journal files, and SYSIDMS file provided during installation.

You can tailor the following IDMSLBLS procedure (provided at installation) to reflect the filenames and definitions in use at your site. Reference IDMSLBLS as shown in the previous z/VSE JCL job stream.

```
* -------- LIBDEFS --------
// LIBDEF  *,SEARCH=idmslib.sublib
// LIBDEF  *,CATALOG=user.sublib
/*  ------------------------- LABELS -------------------------
// DLBL    idmslib,'idms.library',2099/365
// EXTENT  ,nnnnnn,,,ssss,1500
// DLBL    dccat,'idms.system.dccat',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,31
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    dccatl,'idms.system.dccatlod',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,6
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    dccatx,'idms.system.dccatx',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,11
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    dcdml,'idms.system.ddldml',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,101
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    dclod,'idms.system.ddldclod',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,21
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    dclog,'idms.system.ddldclog',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,401
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    dcrun,'idms.system.ddldcrun',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,68
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    dcscr,'idms.system.ddldcscr',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,135
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    dcmsg,'idms.sysmsg.ddldcmsg',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,201
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    dclscr,'idms.sysloc.ddlocscr',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,6
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    dirldb,'idms.sysdirl.ddldml',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,201
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    dirllod,'idms.sysdirl.ddldclod',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,2
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    empdemo,'idms.empdemo1',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,11
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    insdemo,'idms.insdemo1',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,6
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    orgdemo,'idms.orgdemo1',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,6
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    empldem,'idms.sqldemo.empldemo',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,11
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    infodem,'idms.sqldemo.infodemo',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,6
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    projdem,'idms.projseg.projdemo',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,6
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    indxdem,'idms.sqldemo.indxdemo',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,6
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    sysctl,'idms.sysctl',2099/365,SD
// EXTENT  SYSnnn,nnnnnn,,,ssss,2
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    secdd,'idms.sysuser.ddlsec',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,26
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    dictdb,'idms.appldict.ddldml',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,51
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    dloddb,'idms.appldict.ddldclod',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,51
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    sqldd,'idms.syssql.ddlcat',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,101
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    sqllod,'idms.syssql.ddlcatl',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,51
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    sqlxdd,'idms.syssql.ddlcatx',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,26
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    asfdml,'idms.asfdict.ddldml',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,201
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    asflod,'idms.asfdict.asflod',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,401
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    asfdata,'idms.asfdict.asfdata',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,201
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    ASFDEFN,'idms.asfdict.asfdefn',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,101
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    j1jrnl,'idms.j1jrnl',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,54
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    j2jrnl,'idms.j2jrnl',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,54
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    j3jrnl,'idms.j3jrnl',2099/365,DA
// EXTENT  SYSnnn,nnnnnn,,,ssss,54
// ASSGN   SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL    SYSIDMS,'#SYSIPT',0,SD
/+
/*
```

idmslib.sublib

Name of the sublibrary within the library containing CA IDMS modules

user.sublib

Name of the sublibrary within the library containing user modules

idmslib

Name of the file containing CA IDMS modules

idms.library

ID associated with the file containing CA IDMS modules

SYSnnn

Logical unit of the volume for which the extent is effective

nnnnnn

Volume serial identifier of appropriate disk volume

ssss

Starting track (CKD) or block (FBA) of disk extent

dccat

Filename of the system dictionary catalog (DDLCAT) area

idms.system.dccat

ID of the system dictionary catalog (DDLCAT) area

dccatl

Filename of the system dictionary catalog load (DDLCATLOD) area

idms.system.dccatlod

ID of the system dictionary catalog load (DDLCATLOD) area

dccatx

Name of the system dictionary catalog index (DDLCATX) area

idms.system.dccatx

ID of the system dictionary catalog index (DDLCATX) area

dcdml

Name of the system dictionary definition (DDLDML) area

idms.system.ddldml

ID of the system dictionary definition (DDLDML) area

dclod

Name of the system dictionary definition load (DDLDCLOD) area

idms.system.ddldclod

ID of the system dictionary definition load (DDLDCLOD) area

dclog

Name of the system log area (DDLDCLOG) area

idms.system.ddldclog

ID of the system log (DDLDCLOG) area

dcrun

Name of the system queue (DDLDCRUN) area

idms.system.ddldcrun

ID of the system queue (DDLDCRUN) area

dcscr

Name of the system scratch (DDLDCSCR) area

idms.system.ddldcscr

ID of the system scratch (DDLDCSCR) area

dcmsg

Name of the system message (DDLDCMSG) area

idms.sysmsg.ddldcmsg

ID of the system message (DDLDCMSG) area

dclscr

Name of the local mode system scratch (DDLOCSCR) area

idms.sysloc.ddlocscr

ID of the local mode system scratch (DDLOCSCR) area

dirldb

Name of the IDMSDIRL definition (DDLDML) area

idms.sysdirl.ddldml

ID of the IDMSDIRL definition (DDLDML) area

dirllod

Name of the IDMSDIRL definition load (DDLDCLOD) area

idms.sysdirl.dirllod

ID of the IDMSDIRL definition load (DDLDCLOD) area

empdemo

Name of the EMPDEMO area

idms.empdemo1

ID of the EMPDEMO area

insdemo

Name of the INSDEMO area

idms.insdemo1

ID of the INSDEMO area

orgdemo

Name of the ORGDEMO area

idms.orgdemo1

ID of the ORDDEMO area

empldem

Name of the EMPLDEMO area

idms.sqldemo.empldemo

ID of the EMPLDEMO area

infodem

Name of the INFODEMO area

idms.sqldemo.infodemo

ID of the INFODEMO area

projdem

Name of the PROJDEMO area

idms.projseg.projdemo

ID of the PROJDEMO area

indxdem

Name of the INDXDEMO area

idms.sqldemo.indxdemo

ID of the INDXDEMO area

sysctl

Name of the SYSCTL file

idms.sysctl

ID of the SYSCTL file

secdd

Name of the system user catalog (DDLSEC) area

idms.sysuser.ddlsec

ID of the system user catalog (DDLSEC) area

dictdb

Name of the application dictionary definition area

idms.appldict.ddldml

ID of the application dictionary definition (DDLDML) area

dloddb

Name of the application dictionary definition load area

idms.appldict.ddldclod

ID of the application dictionary definition load (DDLDCLOD) area

sqldd

Name of the SQL catalog (DDLCAT) area

idms.syssql.ddlcat

ID of the SQL catalog (DDLCAT) area

sqllod

Name of the SQL catalog load (DDLCATL) area

idms.syssql.ddlcatl

ID of SQL catalog load (DDLCATL) area

sqlxdd

Name of the SQL catalog index (DDLCATX) area

idms.syssql.ddlcatx

ID of the SQL catalog index (DDLCATX) area

asfdml

Name of the asf dictionary definition (DDLDML) area

idms.asfdict.ddldml

ID of the asf dictionary definition (DDLDML) area

asflod

Name of the asf dictionary definition load (ASFLOD) area

idms.asfdict.asflod

ID of the asf dictionary definition load (ASFLOD) area

asfdata

Name of the asf data (ASFDATA) area

idms.asfdict.asfdata

ID of the asf data area (ASFDATA) area

ASFDEFN

Name of the asf data definition (ASFDEFN) area

idms.asfdict.asfdefn

ID of the asf data definition area (ASFDEFN) area

j1jrnl

Name of the first disk journal file

idms.j1jrnl

ID of the first disk journal file

j2jrnl

Name of the second disk journal file

idms.j2jrnl

ID of the second disk journal file

j3jrnl

Name of the third disk journal file

idms.j3jrnl

ID of the third disk journal file

SYSIDMS

Name of the SYSIDMS parameter file

CMS Commands

This section provides the CMS commands to run the schema and subschema compilers (under the central version and in local mode).

Schema Compiler

IDMSCHEM—Central Version IDMSCHEM (CMS)

```
FILEDEF SYSLST PRINTER
FILEDEF SYSPCH DISK syspch output a (RECFM F LRECL 80
FILEDEF SYSIPT schema ddl a (RECFM F LRECL ppp BLKSIZE nnn
FILEDEF SYSIDMS DISK syidms parms a (RECFM F LRECL ppp BLKSIZE nnn
FILEDEF sysctl DISK sysctl idms a
EXEC IDMSFD
OSRUN IDMSCHEM
```

Note: Include the SYSPCH statement only if the DDL specifies PUNCH TO SYSPCH.

syspch output a

File name, type, and mode of the output punch file

schema ddl a

File name, type, and mode of the file that contains the schema DDL statements

ppp

Record length of file

nnn

Block size of file

sysidms parms a

File name, type, and mode of the file that contains the SYSIDMS parameters

sysctl

File name of the SYSCTL file

sysctl idms a

File name, type, and mode of the SYSCTL file

IDMSFD

Exec which defines all FILEDEFs, TXTLIBs, and LOADLIBs required by the system

IDMSCHEM—Local Mode

To execute the schema compiler in local mode, specify local mode in one of the following ways:

- Link IDMSCHEM with an IDMSOPTI program that specifies local execution mode
- Specify *LOCAL* as the first input parameter in the SYSIPT file
- Modify the OSRUN command:

  ```
  OSRUN IDMSCHEM PARM='*LOCAL*'
  ```

  Note: This option is valid only if the OSRUN command is issued from a System Product interpreter or an EXEC2 file.

Creating the SYSIPT File

To create the SYSIPT file, enter these CMS commands:

```
XEDIT sysipt data a (NOPROF
INPUT
 .
 .
 .
Schema source statements
 .
 .
 .
FILE
```

Editing the SYSIPT File

To edit the SYSIDMS parameter file, enter these CMS commands:

```
XEDIT sysidms parms a (NOPROF
INPUT
 .
 .
 .
SYSIDMS parameters
 .
 .
 .
FILE
```

Subschema Compiler

IDMSUBSC—Central Version IDMSUBSC (CMS)

```
FILEDEF SYSLST PRINTER
FILEDEF SYSPCH DISK syspch output a (RECFM F LRECL 80
FILEDEF SYSIPT subsch ddl a (RECFM F LRECL ppp BLKSIZE nnn
FILEDEF SYSIDMS DISK syidms parms a (RECFM F LRECL ppp BLKSIZE nnn
FILEDEF sysctl DISK sysctl idms a
EXEC IDMSFD
OSRUN IDMSUBSC
```

Note: Include the SYSPCH statement only if the DDL specifies PUNCH TO SYSPCH.

subsch ddl a

File name, type, and mode of the file that contains the subschema DDL statements

syspch output a

File name, type, and mode of the output punch file

ppp

Record length of file

nnn

Block size of file

sysidms parms a

File name, type, and mode of the file that contains the SYSIDMS parameters

sysctl

File name of the SYSCTL file

sysctl idms a

File name, type, and mode of the SYSCTL file

IDMSFD

Exec which defines all FILEDEFs, TXTLIBs, and LOADLIBs required by the system

IDMSUBSC—Local Mode

To execute the subschema compiler in local mode, specify local mode in one of the following ways:

- Link IDMSUBSC with an IDMSOPTI program that specifies local execution mode
- Specify *LOCAL* as the first input parameter in the SYSIPT file
- Modify the OSRUN command:

  ```
  OSRUN IDMSUBSC PARM='*LOCAL*'
  ```

  Note: This option is valid only if the OSRUN command is issued from a System Product interpreter or an EXEC2 file.

Note: For more information about creating a SYSIPT file or editing the SYSIDMS file, see the information provided with the IDMSCHEM jobstream.

Appendix E: System Record Types

System Record Types for Space Management

CA IDMS/DB maintains the following nine system record types for space management:

Type	Record ID	Description
SR1	1	Participates as owner in the system-owned CALC set; members are all user record types with a storage mode of CALC; occurs once for each page in a standard database area as bytes 5 through 16 in the header
SR2	2	Replaces records relocated by the RESTRUCTURE, and the migration utility (RHDCMIG1 and RHDCMIG2), and SQL processing following the addition of a column to a table; eight bytes in length
SR3	3	Identifies a user record as having been relocated; the actual user-designated record identification can be found in the relocated record's corresponding SR2 record
SR4	4	Identifies fragments of variable-length records; the actual user-designated record identification can be found in the line index of the root portion of the record
SR5	5	Holds the area-level synchronization stamp for SQL-defined segments and acts as an owner for the table-level synchronization stamp records (SR9s)
SR6	6	Appears in the subschema tables for excluded owner or member record definitions in set relationships; never occurs in the database
SR7	7	Participates as owner in an index; stores CALC under the indexed set's name; occurs once for each indexed set in the database that does not have a user-defined owner record (for details, see Chapter 20, Two-phase Commit Processing")
SR8	8	Contains index entries that point to lower level SR8 records or to an indexed set's member database record occurrences; chained by next, prior, and owner pointers to the owner record occurrence of an indexed set (for details, see Chapter 20, "Two-phase Commit Processing")

Type	Record ID	Description
SR9	9	Holds the table identifier and synchronization stamp for each table in the area

Appendix F: User-Exit Program for Schema and Subschema Compiler

This section contains the following topics:

Overview

This appendix presents the procedures for coding a user-exit program, which is called by the schema compiler and subschema compiler to:

- Perform security checks
- Enforce entity-occurrence naming conventions
- Maintain an audit trail of dictionary activity

A common user-exit program can be coded to be shared by the schema compiler and subschema compiler, or a specialized user-exit program can be coded for each or for only one of the compilers.

The rules and procedures governing the user-exit program are the same for all compilers that use it.

When a User Exit is Called

The user-exit module is called by the applicable compiler when it encounters any of these four points:

- **SIGNON/SIGNOFF/COMMIT**

  After the signon procedure is complete and the compiler's security checks have been passed, or immediately after signoff or COMMIT processing.

- **Major command**

  After an ADD, MODIFY, DELETE, DISPLAY or PUNCH request has been issued. The program is invoked just after the applicable compiler has identified the entity that is the object of the request and has successfully checked authorization requirements. Object entities can be any standard schema or subschema entity type.

- **Card image**

  After each input statement (card image) is passed to the user-exit control block after the statement has been:

  – Scanned and printed on the applicable Compiler Activity List

  – Displayed on the terminal

  – Written to the print file (online compiler interface only)

  The administrator can build an audit trail of accesses and updates to the dictionary.

- **End of converse**

  When one of the following occurs, you can perform a termination activity, such as a write-to-log:

  – Pressing Enter during an online compiler session

  – A batch run of the compiler processes a SIGNOFF statement

  – A batch run of the compiler detects an end-of-file condition

Rules for Writing the User-Exit Program

This section describes the rules that apply to writing the user-exit program.

- **Language**

  You can write the user-exit module in any language that supports OS calling conventions. However, it is recommended that you write user-exit modules in Assembler to allow the online compiler to remain reentrant.

  Note: User-exit modules cannot be CA ADS dialogs.

- **Versions**

  You can code and maintain separate versions of user-exit modules for the batch and online compilers, or you can code modules that can be executed both in batch mode and online.

- **Macros**

  The user-exit facility supports all CA IDMS/DC macros for exits to be used with the online compilers. For exits to be used with the batch compilers, the only CA IDMS/DC macros supported are: #WTL, #ABEND, #GETSTG, #FREESTG, #LOAD, and #DELETE; under z/VSE, the only valid form of #DELETE is EPADDR=.

- **Run units**

  You can start a run unit within an exit; however, you should ensure that the run unit does not deadlock with the applicable compiler run unit. If a user-exit run unit will access a dictionary area, the run unit should ready the object area in a retrieval usage mode.

- **Entry points**

  The user exit invoked by each compiler has a unique entry point name.

Compiler Name	Description	User Exit Entry Point
IDMSCHEM	Batch schema compiler	SCHEXITB
IDMSCHDC	Online schema compiler	SCHEXITO
IDMSUBSC	Batch subschema compiler	SUBEXITB
IDMSUBDC	Online subschema compiler	SUBEXITO

  Although each exit has a unique entry point name, you can use the same exit code for more than one compiler by assigning multiple entry point names to the same set of code.

- **Enabling a compiler exit**

  To enable a user exit for the schema or subschema compilers, link your exit module with IDMSUXIT.

  Note: For more information on how to enable user exits by linking them with IDMSUXIT, refer to the "User Exits" section in the *CA IDMS Systems Operations Guide*.

- **Interface**

  User exits written in COBOL to run under the applicable online compiler require a user-exit interface, written in Assembler with an entry point appropriate to the compiler for which it is to be invoked. This interface should issue a #LINK to the COBOL program (with an entry point other than IDMSCHDC or IDMSUBDC) to isolate it from IDMSCHDC or IDMSUBDC, which is storage-protected.

- **Register conventions**

  User-exit modules are called using the following OS register conventions:

R15	Entry point of module
R14	Return address
R13	18 fullword SAVEAREA
R1	Fullword parameter list

- **Parameters 3 and 4**

  For all four types of user exits, parameter 1 points to a user-exit control block and parameter 2 points to a SIGNON element block. The information addressed in parameters 3 and 4 varies based on the type of user exit, as follows:

  - For the **SIGNON/SIGNOFF/COMMIT** and **end-of-conversation** exits, parameter 3 points to a SIGNON block.
  - For the **major command user exit**, parameter 3 points to an entity control block.
  - For the **card-image** user exit, parameter 3 points to a card-image control block.
  - For **all user exits except the card-image user exit**, parameter 4 is reserved for use by the applicable compiler and should be defined as a PIC X(80) field in the user-exit module.
  - For the **card-image user exit**, parameter 4 points to the input card image, which is defined as a PIC X(80) field.

  The user-exit control blocks are described separately later in this appendix.

- **Information modification**

  With the exception of the fields identified within the user-exit control block, a user-exit module should not modify any of the information passed.

- **Return codes**

  On return from a user-exit module, you must set a return code and, optionally, specify a message ID and message text to be issued by the applicable compiler, as follows:

Code	Compiler Action
0	No message is issued; compiler continues with normal processing.
1	An informational message is issued; compiler continues with normal processing.
4	A warning message is issued; compiler continues with normal processing.
8	An error message is issued; compiler initiates error processing.

Control Blocks and Sample User-Exit Programs

This section presents the formats of these five control blocks:

- User-exit control block
- SIGNON element block
- SIGNON block
- Entity control block
- Card-image control block

User-Exit Control Block

The following table shows how to define the user-exit control block:

Field	Usage	Size	Picture	Description
1	Char	8	X(8)	Compiler name: IDMSCHEM or IDMSUBSC
2	Char	8	X(8)	Compiler start date: mm/dd/yy
3	Char	8	X(8)	Compiler start time: hhmmssmm
4	Binary	4	S9(8) COMP	User field initialized to 0 (for use by reentrant modules as a pointer to their work area)
5	Binary	4	S9(8) COMP	User return code (described next)
6	Char	8	X(8)	Message ID returned by user, in the range DC900000 through DC999999, or any 6-digit number; blank if no message is returned
7	Char	80	X(80)	Message text returned by user

SIGNON Element Block

The following table shows how to define the SIGNON element block:

Field	Usage	Size	Picture	Description
1	Binary	1	X	Length of user ID for #WTLs (not addressable by COBOL)
2	Char	32	X(32)	SIGNON user ID

SIGNON Block

The following table shows how to define the SIGNON block.

Field	Usage	Size	Picture	Description
1	Char	16	X(16)	SIGNON, SIGNOFF, COMMIT or END-OF-CONVERSE statement
2	Char	8	X(8)	SIGNON dictionary name
3	Char	8	X(8)	SIGNON node name
4A	CHAR	32	X(32)	User ID
5	Binary	2	S9(4)	DDLDML area usage mode: 36=UPDATE; 38=PROTECTED UPDATE; 37=RETRIEVAL
6	Binary	2	S9(4)	DDLDCLOD area usage mode
7	Binary	2	S9(4)	DDLDCMSG area usage mode
8	Binary	10	X(10)	Reserved

Note: Each bit in flag 0 and flag 1 must be tested separately. More than one bit may be on at any one time.

Entity Control Block

The following table shows how to define the entity control block.

Field	Usage	Size	Picture	Description
1	Char	16	X(16)	Major command (ADD, MODIFY, DELETE, DISPLAY, or PUNCH)
2	Char	32	X(32)	Entity type
3	Char	40	X(40)	Entity occurrence
4	Binary	2	S9(4)	Entity version number or number of records requested
5	Char	64	X(64)	Additional Qualifier
6	Char	32	X(32)	PREPARED BY user ID
7	Char	32	X(32)	REVISED BY user ID

Card-image Control Block

The following table shows how to define the card-image control block:

Field	Usage	Size	Picture	Description
1	Char	16	X(16)	Compiler 'CARD IMAGE' command
2	Binary	2	S9(8)	Input low-card column
3	Binary	2	S9(8)	Input high-card column

Sample User-Exit Program for Schema and/or Subschema Compilers

The following sample user-exit program can be used to enforce naming conventions for elements in the batch and online versions of the schema and subschema compilers. The source code for this program can be found in the installation source library under member name IDDSUXIT.

```
********************************************************************
IDDUXIT  TITLE 'NAMING CONVENTION CHECKER'
********************************************************************
*
*
*  PROGRAM NAME : IDDUXIT
*
*  DATE         : 03/01/96
*
*
*  DESCRIPTION  : THIS IS AN EXAMPLE OF A USER EXIT.  THIS PROGRAM
*                 SHOWS HOW A SHOP COULD CHECK THE ENTITY NAMES FOR
*                 A SHOP STANDARD.  ANY VIOLATIONS OF THE NAMING
*                 CONVENTION ARE TREATED AS AN ERROR AND THE ACTION
*                 (ADD, MOD, DEL) IS NOT ALLOWED.
```

```
*********************************************************************
IDDUXIT  CSECT
         #REGEQU
         ENTRY SCHEXITO
SCHEXITO DS    0H                    Online Schema compiler entry
         ENTRY SCHEXITB
SCHEXITB DS    0H                    Batch Schema compiler entry
         ENTRY SUBEXITO
SUBEXITO DS    0H                    Online Subschema compiler entry
         ENTRY SUBEXITB
SUBEXITB DS    0H                    Batch Subschema compiler entry
*********************************************************************
*        SET UP ADDRESSABILITY                                      *
*********************************************************************
         STM   R14,R12,12(R13)      SAVE CALLERS REGISTERS
         LR    R12,R15
         USING IDDUXIT,R12
         L     R4,12(R1)             GET THE
         L     R3,8(R1)                 CORRECT
         L     R2,4(R1)                    PARAMETER
         L     R1,0(R1)                       ADDRESSES
*
IDDUXITR DS    0H                    BASE THE CONTROL BLOCKS
*
         USING UXITCB,R1             USER EXIT CONTROL BLOCK
         MVC   UXITRCDE,F0           ZERO OUT THE RETURN CODE
         MVC   UXITMID(8),BLANKS     BLANK OUT THE MESSAGE ID
         MVC   UXITMTXT(80),BLANKS   BLANK OUT THE MESSAGE
*
```

```
**********************************************************************
*        INTERROGATE THE MAJOR COMMAND                               *
**********************************************************************
*
         SPACE
UXIENTY  EQU   *
         USING UXITECB,R3            ENTITY CONTROL BLOCK
*
         CLC   UXITEVRB,UXICSON        IS IT A SIGNON?
         BE    USIGNON                   YES, CHECK THE USER NAME
*
         CLC   UXITEVRB,UXICARD        IS IT A CARD IMAGE EXIT?
         BE    UCARD                     YES, CHECK THE CARD
*
         CLC   UXITEVRB,UXICADD        IS IT AN ADD?
         BE    UXIECHK                   YES, CHECK THE ENTITY-NAME
*
         CLC   UXITEVRB,UXICMOD        IS IT A MODIFY?
         BE    UXIECHK                   YES, CHECK THE ENTITY-NAME
*
         CLC   UXITEVRB,UXICDEL        IS IT A DELETE?
         BE    UXIECHK                   YES, CHECK THE ENTITY-NAME
*                                        NO
         MVC   UXITMID(8),ELSEID      MOVE IN 'ELSE' MESSAGE ID
         MVC   UXITMTXT(80),ELSEMSG   MOVE IN 'ELSE' MESSAGE
         B     UXIEBYE
*
```

```
**********************************************************************
*        CHECK THE CARD IMAGE                                        *
**********************************************************************
*
         SPACE
UCARD    EQU   *
*
         MVC   UXITMID(8),CARDID     FILL IN THE MESSAGE ID
         MVC   UXITMTXT(80),CARDMSG  FILL IN THE MESSAGE TEXT
         B     UXIEBYE               BACK TO THE COMPILER
*
**********************************************************************
*        CHECK THE USER NAME FOR ME                                  *
**********************************************************************
*
         SPACE
USIGNON  EQU   *
*
         USING UXITSEB,R2            SIGNON ELEMENT BLOCK
         USING UXITSB,R3             SIGNON BLOCK
*
         CLC  UXITUSER(3),WHOME      IS IT ME
         BE   UXIEDC                 YES GO CHECK FOR DC NAME
*                                    NO, GO TO JAIL, GO DIRECTLY TO
*                                    JAIL, DO NOT PASS GO DO NOT
USNAME   EQU   *                     COLLECT $200.
         MVC   UXITRCDE,F8           FILL IN THE RETURN CODE
         MVC   UXITMID(8),NOSNID     FILL IN THE MESSAGE ID
         MVC   UXITMTXT(80),NOSNMSG  FILL IN THE MESSAGE TEXT
         B     UXIEBYE               BACK TO THE COMPILER
*
UXIEDC   EQU   *
         TM    UXITFLG1,UXIT1DC      ARE WE RUNNING DC
         BZ    UXIEBYE               NO, SKIP DC ID CHECK
*
         CLC   UXITUSER,UXITIUSR     IS THE USER THE SAME AS DC
         BE    UXIEBYE               YES, OK LET IT PASS
*                                    NO, DON'T LET THEM SIGNON
         MVC   UXITRCDE,F8           FILL IN THE RETURN CODE
         MVC   UXITMID(8),NODCID     FILL IN THE MESSAGE ID
         MVC   UXITMTXT(80),NODCMSG  FILL IN THE MESSAGE TEXT
         B     UXIEBYE               BACK TO THE COMPILER
*
```

```
*********************************************************************
*        CHECK THE ENTITY-NAME FOR VALID NAMING CONVENTION          *
*********************************************************************
*
         SPACE
UXIECHK  EQU   *
         USING UXITECB,R3            ENTITY CONTROL BLOCK
*
         CLC   UXITENME(3),NAMECHK   DOES THE NAME FOLLOW THE RULES?
         BE    UXIEBYE                 YES, LET THIS ONE PASS.
*                                      NO, RETURN AN ERROR
*
         MVC   UXITRCDE,F8           FILL IN THE RETURN CODE
         MVC   UXITMID(8),NONOID     FILL IN THE MESSAGE ID
         MVC   UXITMTXT(80),NONOMSG  FILL IN THE MESSAGE TEXT
*
********************************************************************
*        RETURN BACK TO THE COMPILER                                        *
********************************************************************
*
         SPACE
UXIEBYE  EQU   *
         LM    R14,R12,12(R13)       RELOAD CALLER'S REGISTERS
         BR    R14                   RETURN TO CALLER
         EJECT
```

```
********************************************************************
*        CONSTANTS AND LITERALS                                    *
********************************************************************
UXICADD  DC    CL16'ADD             '
UXICMOD  DC    CL16'MODIFY          '
UXICDEL  DC    CL16'DELETE          '
UXICSON  DC    CL16'SIGNON          '
UXICARD  DC    CL16'CARD IMAGE      '
NAMECHK  DC    CL3'XYZ'
WHOME    DC    CL3'XYZ'
WKLEN    DC    F'100'
NONOID   DC    CL8'DC999001'
NONOMSG  DC    CL80'NAMING CONVENTION VIOLATED - ACTION NOT ALLOWED'
NOSNID   DC    CL8'DC999002'
NOSNMSG  DC    CL80'SIGNON ERROR - USER NOT ALLOWED ACCESS'
NODCID   DC    CL8'DC999003'
NODCMSG  DC    CL80'SIGNON ERROR - USER NAME NOT DC USER NAME'
CARDID   DC    CL8'DC999004'
CARDMSG  DC    CL80'MESSAGE PRODUCED BY CARD IMAGE EXIT          '
ELSEID   DC    CL8'DC999005'
ELSEMSG  DC    CL80'MESSAGE PRODUCED BY CARD IMAGE EXIT          '
BLANKS   DC    CL80' '
F0       DC    F'0'               NORMAL RETURN CODE - NO ERRORS
F2       DC    F'1'               INFORMATION MESSAGE
F4       DC    F'4'               WARNING MESSAGE
F8       DC    F'8'               ERROR MESSAGE
*
********************************************************************
*        USER EXIT CONTROL BLOCK                                   *
********************************************************************
UXITCB   DSECT
UXITCPLR DS    CL8           COMPILER NAME 'IDMSCHEM' OR 'IDMSUBSC'
UXITDATE DS    CL8           COMPILER START DATE MM/DD/YY
UXITTIME DS    CL8           COMPILER START TIME HHMMSSMM
UXITWORK DS    F             USER FULLWORD INITIALIZED TO 0
UXITRCDE DS    0F            RETURN CODE RETURNED BY USER
         DS    XL3           UNUSED
UXITRC   DS    X
UXITRC00 EQU   X'00'              NORMAL RETURN CODE - NO ERRORS
UXITRC01 EQU   X'01'              INFORMATION MESSAGE
UXITRC04 EQU   X'04'              WARNING MESSAGE
UXITRC08 EQU   X'08'              ERROR MESSAGE
UXITMID  DS    CL8           USER MESSAGE ID RETURNED BY USER
UXITMTXT DS    CL80          USER MESSAGE TEXT RETURNED BY USER
UXITCBLN EQU   *-UXITCB      USER EXIT CONTROL BLOCK LENGTH
*
```

```
********************************************************************
*        USER EXIT SIGNON ELEMENT BLOCK                            *
********************************************************************
UXITSEB  DSECT
UXITIDLN DS    X             LENGTH OF USERID FOR #WTL'S
UXITUSER DS    CL32          USER ID
         DS    0A            ROUND UP TO FULLWORD
UXITSNLN EQU   *-UXITSEB     LENGTH OF SIGNON ELEMENT
*
********************************************************************
*        USER EXIT SIGNON BLOCK                                    *
********************************************************************
UXITSB   DSECT
UXITTYPE DS    CL16          VERB
UXITDICT DS    CL8           DICTIONARY NAME
UXITNODE DS    CL8           NODE NAME
UXITIUSR DS    CL32          USER ID

UXITIPSW DS    CL8           USER'S PASSWORD
UXITFLG0 DS    CL1           ENVIRONMENT FLAG
UXIT0DOS EQU   X'80'           COMPILER RUNNING UNDER z/VSE
UXIT0MEN EQU   X'40'           RUNNING UNDER 'MENU' MODE
UXITFLG1 DS    CL1           ENVIRONMENT FLAG
UXIT1LCL EQU   X'80'           RUNNING IN INTERNAL SUBROUTINE MODE
UXIT1DC  EQU   X'40'           COMPILER RUNNING UNDER DC
         DS    CL2           RESERVED FOR FUTURE FLAGS
         DS    CL20          RESERVED
UXITDMLM DS    H             DDLDML USAGE MODE
*                                36=UPDATE
*                                37=PROTECTED UPDATE
*                                38=RETRIEVAL
UXITLODM DS    H             DDLDCLOD USAGE MODE
UXITMSGM DS    H             DDLDCMSG USAGE MODE
         DS    CL10          RESERVED
UXITSLEN EQU   *-UXITSB      LENGTH OF USER EXIT SIGNON BLOCK
*
```

```
********************************************************************
*        USER EXIT ENTITY CONTROL BLOCK                            *
********************************************************************
UXITECB  DSECT
UXITEVRB DS    CL16          VERB
UXITENTY DS    CL32          ENTITY-TYPE
UXITENME DS    CL40          ENTITY NAME
UXITEVER DS    H             VERSION
UXITEADQ DS    CL64          ADDITIONAL QUALIFIER
UXITPREP DS    CL32          PREPARED BY USER NAME
UXITREV  DS    CL32          REVISED BY USER NAME
UXITELEN EQU   *-UXITECB     LENGTH OF USER EXIT ENTITY CONTROL BLK
*
********************************************************************
*        END OF EXIT                                               *
********************************************************************
         END
```

Appendix G: Quick Reference Information

This section contains quick reference information which can be useful when performing administration tasks in the database.

This section contains the following topics:

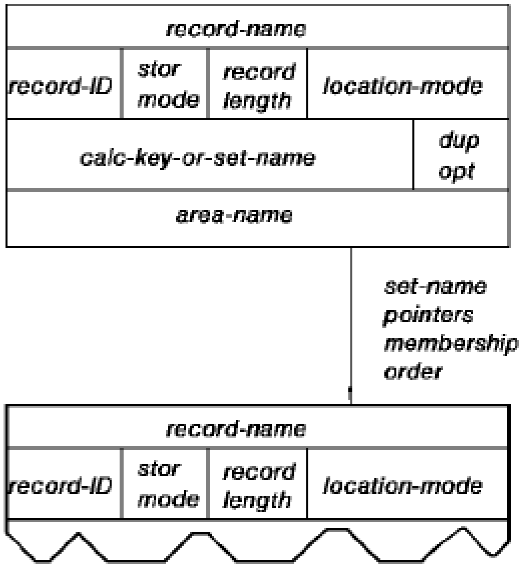
Editing Commands

Edit Command	Command Format
COPY	Copy a single line: %C Copy this line and the next n-1 lines: %Cn Copy a block of lines: %CB (on the first line of the block) %CE (on the last line of the block) You follow a COPY command with an AFTER or BEFORE command.
DELETE	Delete a single line: %D Delete this line and the next n-1 lines: %Dn Delete a block of lines: %DB (on the first line of the block) %DE (on the last line of the block)
MOVE	Move a single line: %M Move this line and the next n-1 lines: %Mn Move a block of lines: %MB (on the first line of the block) %ME (on the last line of the block) You follow a MOVE command with an AFTER or BEFORE command.

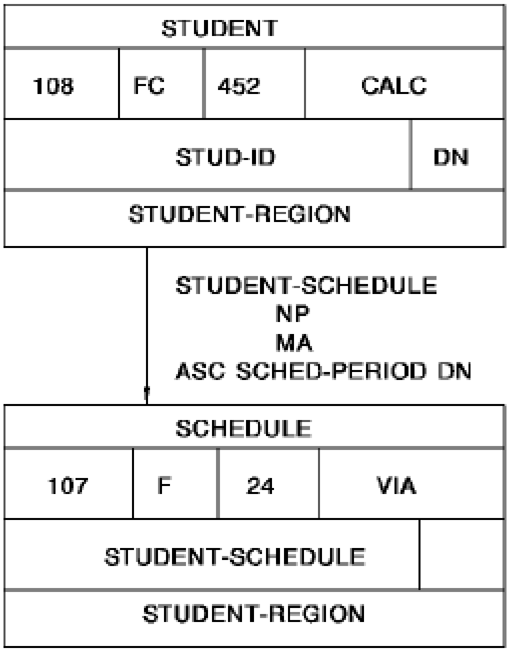
Edit Command	Command Format
REPEAT	Repeat a single line: %R Repeat this line and the next n-1 lines: %Rn Repeat a block of lines: %RB (on the first line of the block) %RE (on the last line of the block)
AFTER	Place a COPY or MOVE block after the line on which this command is placed: %A
BEFORE	Place a COPY or MOVE block before the line on which this command is placed: %B
TOP	Reposition the work file so that the line on which the command appears becomes the top line on the screen: %T

Record-Set Representation

Format



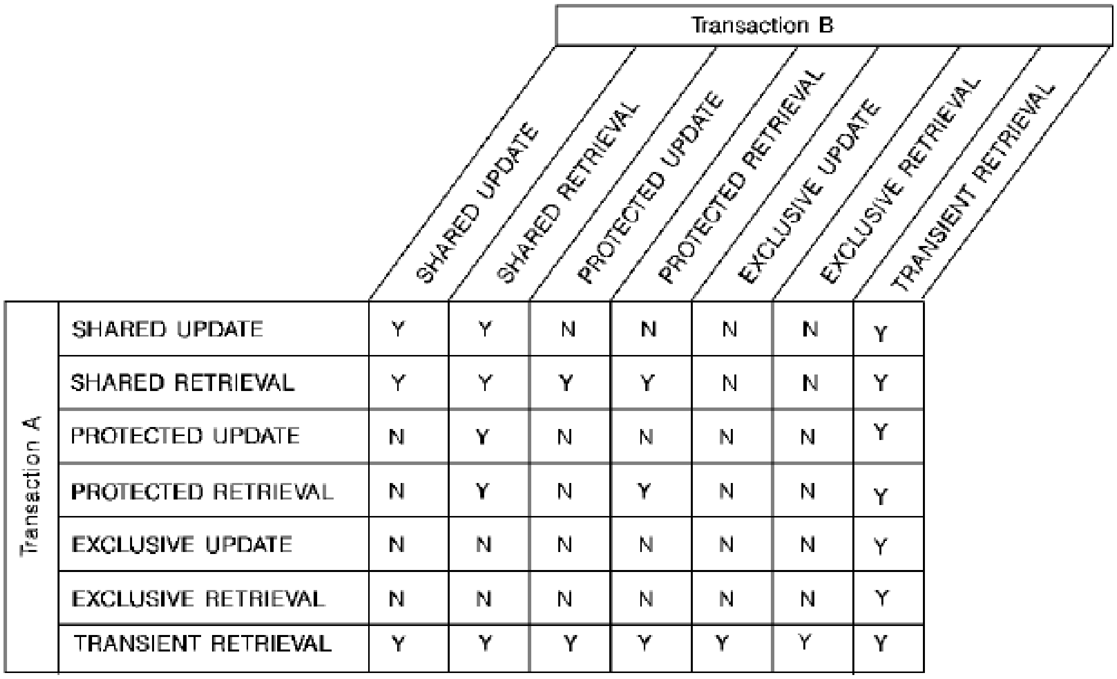
Example



stor mode	= F, FC, V, or VC
location-mode	= CALC, VIA, DIRECT, VSAM, or VSAM CALC
dup opt	= DN, DF, DL, or DU
pointers	= N, NP, NO, NPO, I, or IO
membership	= MA, MM, OA, or OM
order	
for unsorted sets	= FIRST, LAST, NEXT, or PRIOR
for sorted sets	= ASC or DES/ *sort-key/dup opt for sort-key*

Lock Management

Ready Mode Compatibility



		Transaction B						
		SHARED UPDATE	SHARED RETRIEVAL	PROTECTED UPDATE	PROTECTED RETRIEVAL	EXCLUSIVE UPDATE	EXCLUSIVE RETRIEVAL	TRANSIENT RETRIEVAL
Transaction A	SHARED UPDATE	Y	Y	N	N	N	N	Y
	SHARED RETRIEVAL	Y	Y	Y	Y	N	N	Y
	PROTECTED UPDATE	N	Y	N	N	N	N	Y
	PROTECTED RETRIEVAL	N	Y	N	Y	N	N	Y
	EXCLUSIVE UPDATE	N	N	N	N	N	N	Y
	EXCLUSIVE RETRIEVAL	N	N	N	N	N	N	Y
	TRANSIENT RETRIEVAL	Y	Y	Y	Y	Y	Y	Y

Lock Resource ID Format

Resource Type	Bytes 1-4	Bytes 5-8
Dbkey	X'nnnn00xx	Dbkey
Page	X'nnnn10xx'	Page number
Space on a page	X'nnnn20xx'	Page number
Area	X'nnnn80xx	Area low page number
Area (transient retrieval)	X'nnnnC0xx'	Area low page number

Runtime Error-Status Codes

Major DB Status Codes

Major Code	Database Function
00	Any DML statement
01	FINISH
02	ERASE
03	FIND/OBTAIN
05	GET
06	KEEP
07	CONNECT
08	MODIFY
09	READY
11	DISCONNECT
12	STORE
14	BIND
15	ACCEPT
16	IF
17	RETURN
18	COMMIT
19	ROLLBACK
20	LRF requests

Minor DB Status Codes

Minor Code	Database Function Status
00	Combined with a major code of 00, this code indicates successful completion of the DML operation. Combined with a nonzero major code, this code indicates that the DML operation was not completed successfully due to central version causes, such as time-outs and program checks.

Minor Code	Database Function Status
01	An area has not been readied. When this code is combined with a major code of 16, an IF operation has resulted in a valid false condition.
02	Either the db-key used with a FIND/OBTAIN DB-KEY statement or the direct db-key suggested for a STORE is not within the page range for the specified record name.
03	Invalid currency for the named record, set, or area. This can only occur when a run unit is sharing a transaction with other database sessions. The 03 minor status is returned if the run unit tries to retrieve or update a record using a currency that has been invalidated because of changes made by another database session that is sharing the same transaction.
04	The occurrence count of a variably occurring element has been specified as either less than zero or greater than the maximum number of occurrences defined in the control element.
05	The specified DML function would have violated a duplicates-not-allowed option for a CALC, sorted, or index set.
06	No currency has been established for the named record, set, or area.
07	The end of a set, area, or index has been reached or the set is empty.
08	The specified record, set, procedure, or LR verb is not in the subschema or the specified record is not a member of the set.
09	The area has been readied with an incorrect usage mode.
10	An existing access restriction or subschema usage prohibits execution of the specified DML function. For LRF users, the subschema in use allows access to database records only. Combined with a major code of 00, this code means the program has attempted to access a database record, but the subschema in use allows access to logical records only.
11	The record cannot be stored in the specified area due to insufficient space.
12	There is no db-key for the record to be stored. This is a system internal error and should be reported.
13	A current record of run unit either has not been established or has been nullified by a previous ERASE statement.
14	The CONNECT statement cannot be executed because the requested record has been defined as a mandatory automatic member of the set.
15	The DISCONNECT statement cannot be executed because the requested record has been defined as a mandatory member of the set.
16	The record cannot be connected to a set of which it is already a member.
17	The transaction manager encountered an error.

Minor Code	Database Function Status
18	The record has not been bound.
19	The run unit's transaction was forced to back out.
20	The current record is not the same type as the specified record name.
21	Not all areas being used have been readied in the correct usage mode.
22	The record name specified is not currently a member of the set name specified.
23	The area name specified is either not in the subschema or not an extent area; or the record name specified has not been defined within the area name specified.
25	No currency has been established for the named set.
26	No duplicates exist for the named record or the record occurrences cannot be found.
28	The run unit has attempted to ready an area that has been readied previously.
29	The run unit has attempted to place a lock on a record that is locked already by another run unit. A deadlock results. Unless the run unit issued either a FIND/OBTAIN KEEP EXCLUSIVE or a KEEP EXCLUSIVE, the run unit is aborted.
30	An attempt has been made to erase the owner record of a nonempty set.
31	The retrieval statement format conflicts with the record's location mode.
32	An attempt to retrieve a CALC/DUPLICATE record was unsuccessful; the value of the CALC field in variable storage is not equal to the value of the CALC control element in the current record of run unit.
33	At least one set in which the record participates has not been included in the subschema.
40	The WHERE clause in an OBTAIN NEXT logical-record request is inconsistent with a previous OBTAIN FIRST or OBTAIN NEXT command for the same record. Previously specified criteria, such as reference to a key field, have been changed. A path status of LR-ERROR is returned to the LRC block.
41	The subschema contains no path that matches the WHERE clause in a logical-record request. A path status of LR-ERROR is returned to the LRC block.
42	An ON clause included in the path by the DBA specified return of the LR-ERROR path status to the LRC block; an error has occurred while processing the LRF request.

Minor Code	Database Function Status
43	A program check has been recognized during evaluation of a WHERE clause; the program check indicates that either a WHERE clause has specified comparison of a packed decimal field to an unpacked nonnumeric data field, or data in variable storage or a database record does not conform to its description. A path status of LR-ERROR is returned to the LRC block unless the DBA has included an ON clause to override this action in the path.
44	The WHERE clause in a logical-record request does not supply a key element (sort key, CALC key, or db-key) expected by the path. A path status of LR-ERROR is returned to the LRC block.
45	During evaluation of a WHERE clause, a program check has been recognized because a subscript value is neither greater than 0 nor less than its maximum allowed value plus 1. A path status of LR-ERROR is returned to the LRC block unless the DBA has included an ON clause to override this action in the path.
46	A program check has revealed an arithmetic exception (for example: overflow, underflow, significance, divide) during evaluation of a WHERE clause. A path status of LR-ERROR is returned to the LRC block unless the DBA has included an ON clause to override this action in the path.
53	The subschema definition of an indexed set does not match the indexed set's physical structure in the database.
54	Either the prefix length of an SR51 record is less than zero or the data length is less than or equal to zero.
55	An invalid length has been defined for a variable-length record.
56	An insufficient amount of memory to accommodate the CA IDMS compression/decompression routines is available.
57	A retrieval-only run unit has detected an inconsistency in an index that should cause an 1143 abend, but optional APAR bit 216 has been turned on.
58	An attempt was made to rollback updates in a local mode program. Updates made to an area during a local mode program's execution cannot be automatically rolled out. The area must be manually recovered.
60	A record occurrence type is inconsistent with the set named in the ERROR-SET field in the IDMS communications block. This code usually indicates a broken chain.
61	No record can be found for an internal db-key. This code usually indicates a broken chain.
62	A system-generated db-key points to a record occurrence, but no record with that db-key can be found. This code usually indicates a broken chain.

Minor Code	Database Function Status
63	The DBMS cannot interpret the DML function to be performed. When combined with a major code of 00, this code means invalid function parameters have been passed on the call to the DBMS. For LRF users, a WHERE clause includes a keyword that is longer than the 32 characters allowed.
64	The record cannot be found; the CALC control element has not been defined properly in the subschema.
65	The database page read was not the page requested.
66	The area specified is not available in the requested usage mode.
67	The subschema invoked does not match the subschema object tables.
68	The CICS interface was not started.
69	A BIND RUN-UNIT may not have been issued; the CV may be inactive or not accepting new run units; or the connection with the CV may have been broken due to time out or other factors. When combined with a major code of 00, this code means the program has been disconnected from the DBMS.
70	The database will not ready properly; a JCL error is the probable cause.
71	The page range or page group for the area being readied or the page requested cannot be found in the DMCL.
72	There is insufficient memory to dynamically load a subschema or database procedure.
73	A central version run unit will exceed the MAXERUS value specified at system generation.
74	The dynamic load of a module has failed. If operating under the central version, a subschema or database procedure module either was not found in the data dictionary or the load (core image) library or, if loaded, will exceed the number of subschema and database procedures provided for at system generation.
75	A read error has occurred.
76	A write error has occurred.
77	The run unit has not been bound or has been bound twice. When combined with a major code of 00, this code means either the program is no longer signed on to the subschema or the variable subschema tables have been overwritten.
78	An area wait deadlock has occurred.
79	The run unit has requested more db-key locks than are available to the system.

Minor Code	Database Function Status
80	The target node is either not active or has been disabled.
81	The converted subschema requires specified database name to be in the DBNAME table.
82	The subschema must be named in the DBNAME table.
83	An error has occurred in accessing native VSAM data sets.
87	The owner and member records for a set to be updated are not in the same page group or do not have the same db-key radix.
91	The subschema requires a DBNAME to do the bind run unit.
92	No subschema areas map to DMCL.
93	A subschema area symbolic was not found in DMCL.
94	The specified dbname is neither a dbname defined in the DBNAME table, nor a SEGMENT defined in the DMCL.
95	The specified subschema failed DBTABLE mapping using the specified dbname.

Note: For a complete description of DB runtime status codes, see the chapter "CA IDMS Status Codes" in the *Messages and Codes Guide*.

Major DC Status Codes

Major Code	Function
00	Any DML statement
30	TRANSFER CONTROL
31	WAIT/POST
32	GET STORAGE/FREE STORAGE
33	SET ABEND EXIT/ABEND CODE
34	LOAD/DELETE TABLE
35	GET TIME/SET TIMER
36	WRITE LOG
37	ATTACH/CHANGE PRIORITY
38	BIND/ACCEPT/END TRANSACTION STATISTICS

Major Code	Function
39	ENQUEUE/DEQUEUE
40	SNAP
43	PUT/GET/DELETE SCRATCH
44	PUT/GET/DELETE QUEUE
45	BASIC MODE TERMINAL MANAGEMENT
46	MAPPING MODE TERMINAL MANAGEMENT
47	LINE MODE TERMINAL MANAGEMENT
48	ACCEPT/WRITE PRINTER
49	SEND MESSAGE
50	COMMIT TASK/ROLLBACK TASK/FINISH TASK/WRITE JOURNAL
51	KEEP LONGTERM
58	SVC SEND/RECEIVE

Minor DC Status Codes

Minor Code	Function Status
00	Combined with a major code of 00, this code indicates either successful completion of the DML function or that all tested resources have been enqueued.
01	The requested operation cannot be performed immediately; waiting will cause a deadlock.
02	Either there is insufficient storage in the storage pool or the storage required for control blocks is unavailable.
03	The scratch area ID cannot be found.
04	Either the queue ID (header) cannot be found or a paging session was in progress when a second STARTPAGE command was received (that is, an implied ENDPAGE was processed before this STARTPAGE was executed successfully).
05	The specified scratch record ID or queue record cannot be found.
06	No resource control element (RCE) exists for the queue record; currency has not been established.

Minor Code	Function Status
07	Either an I/O error has occurred or the queue upper limit has been reached.
08	The requested resource is not available.
09	The requested resource is available.
10	New storage has been assigned.
11	A maximum task condition exists.
12	The named task code is invalid.
13	The named resource cannot be found.
14	The requested module is defined as nonconcurrent and is currently in use.
15	The named module has been overlaid and cannot be reloaded immediately.
16	The specified interval control element (ICE) address cannot be found.
17	The record has been replaced.
18	No printer terminals have been defined for the current DC system.
19	The return area is too small; data has been truncated.
20	An I/O, program-not-found, or potential-deadlock status condition exists.
21	The message destination is undefined, the long term ID cannot be found, or a KEEP LONGTERM request was issued by a nonterminal task.
22	A record already exists for the scratch area specified.
23	No storage or resource control element (RCE) could be allocated for the reply area.
24	The maximum number of outstanding replies has been exceeded.
25	An attention interrupt has been received.
26	There is a logical error in the output data stream.
27	A permanent I/O error has occurred.
28	The terminal dial-up line is disconnected.
29	An invalid parameter has been passed in the list set up by the DML processor.
30	The named function has not yet been implemented.
31	An invalid parameter has been passed; the TRB, LRB, or MRB contains an invalid field; or the request is invalid because of a possible logic error in the application program. In a DC-BATCH environment, a possible cause is that the record length specified by the command exceeds the maximum length based on the packet size.

Minor Code	Function Status
32	The derived length of the specified variable storage is negative or zero.
33	Either the named table or the named map cannot be found in the data dictionary load area.
34	The named variable-storage area must be an 01-level entry in the LINKAGE SECTION.
35	A GET STORAGE request is invalid because the LINKAGE SECTION variable has already been allocated.
36	The program either was not defined during system generation or is marked out-of-service.
37	A GET STORAGE operand is invalid because the specified variable storage area is in the WORKING-STORAGE SECTION instead of the LINKAGE SECTION.
38	Either no GET STORAGE operand was specified or the specified LINKAGE SECTION variable has not been allocated.
39	The terminal device being used is out of service.
40	NOIO has been specified but the datastream cannot be found.
41	An IF operation resulted in a valid true condition.
42	The named map does not support the terminal device in use.
43	A line I/O session has been cancelled by the terminal operator.
44	The referenced field does not participate in the specified map; a possible cause is an invalid subscript.
45	An invalid terminal type is associated with the issuing task.
46	A terminal I/O error has occurred.
47	The named area has not been readied.
48	The run unit has not been bound.
49	NOWAIT has been specified but WAIT is required.
50	Statistics are not being kept.
51	A lock manager error occurred during the processing of a KEEP LONGTERM request
52	The specified table is missing or invalid.
53	An error occurred from a user-written edit routine.
54	Either there is invalid internal data or a data conversion error has occurred.
55	The user-written edit routine cannot be found.

Minor Code	Function Status
56	No DFLDS have been defined for the map.
57	The ID cannot be found, is not a long-term permanent ID, or is being used by another run unit.
58	Either the LRID cannot be found, the maximum number of concurrent task threads was exceeded, or an attempt was made to rollback database changes in local mode.
59	An error occurred in transferring the KEEP LONGTERM request to IDMSKEEP
60	The requested KEEP LONGTERM lock id was already in use with a different page group
63	Invalid function parameters have been passed on the call to the DBMS.
64	No detail exists currently for update; no action has been taken. Alternatively, the requested node for a header or detail is either not present or not updated.
68	There are no more updated details to MAP IN or the amount of storage defined for pageable maps at sysgen is insufficient. In the latter case, subsequent MAP OUT DETAIL statements are ignored.
72	No detail occurrence, footer, or header fields exist to be mapped out by a MAP OUT RESUME command, or the scratch record that contains the requested detail could not be accessed. The latter case is a mapping internal error and should be reported.
76	The first screen page has been transmitted to the terminal.
77	Either the program is no longer signed on to the subschema or the variable subschema tables have been overwritten.
80	The target node is either not active or has been disabled.
97	An error was encountered processing a syncpoint request; check the log for details.
98	An unsupported COBOL compiler option (for example, DEBUG) has been specified for an online program or a program running in a batch region has issued a DML verb that is only valid when running online under CA IDMS/DC/UCF.
99	An unexpected internal return code has been received; the terminal device is out of service.

Note: For a complete description of DC runtime status codes, see the chapter "CA IDMS Status Codes" in the *Messages and Codes Guide*.

ERROR-STATUS Condition Names

Code	Condition name	Explanation
0000	DB-STATUS-OK	No error
0307	DB-END-OF-SET	End of set, area, or SPF index
0326	DB-REC-NOT-FOUND	No record found
0001 to 9999	ANY-ERROR-STATUS	Any nonzero status
0000 to 9999	ANY-STATUS	Any status
3101 3201 3401 3901	DC-DEADLOCK	Waiting will cause a deadlock
3202 3402	DC-NO-STORAGE	Insufficient space available
4303	DC-AREA-ID-UNK	ID cannot be found
4404	DC-QUEUE-ID-UNK	Queue header cannot be found
4305 4405	DC-REC-NOT-FOUND	Record cannot be found
3908	DC-RESOURCE-NOT-AVAIL	Resource not available
3909	DC-RESOURCE-AVAIL	Resource is available
3210	DC-NEW-STORAGE	New space allocated
3711	DC-MAX-TASKS	Maximum attached tasks
4317	DC-REC-REPLACED	Record has been replaced
4319 4419 4519 4719	DC-TRUNCATED-DATA	Return area too small; data has been truncated
4525 4625	DC-ATTN-INT	Attention interrupt received
4743	DC-OPER-CANCEL	Session cancelled

Index