CA IDMS™

Database Design 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 ADS™
- CA IDMS™/DB
- CA IDMS™ SQL
- CA IDMS™ Presspack

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 release of this documentation:

- Assigning Segments to Page Groups (see page 254)—Removed indexes from the MPGI restriction

Contents

Chapter 1: Introduction

This section contains the following topics:

Overview

A database is a computer representation of information that exists in the real world. Like a painting, a database tries to imitate reality. Designing a database is an art form, and a successful database bears the mark of a thoughtful, creative designer.

For a given database problem, there may be several solutions. While some designs are clearly better than others, there is no right or wrong design. The structure of your database will therefore be determined not only by the requirements of the business but also by your individual style as a designer. As you develop and refine the design for a database, let your intuition be your guide.

The purpose of creating a database is to satisfy the information requirements of business application programs. Before users can run their application programs, the database administrator (DBA) must design and implement the corporate database. As the DBA or database designer, you are responsible for database design and implementation.

Data models

To design a database, you must develop two different data models:

- The *logical model* describes all corporate information to be maintained in the database. This model represents the way the user perceives the data.
- The *physical model* describes how the data is stored and accessed by the system. The physical design for a database builds on the logical model. During the physical design process, you tailor the logical design to specific application performance requirements and plan the best use of storage resources.

Iterative process

Creating a design for a database is an iterative process. After you have developed the logical and physical models, you need to review the design process and the available documentation with users in your corporation. As users make suggestions for improvement, you should make appropriate changes to the design. Review the design repeatedly until it is acceptable to the user community.

Design Implementation

The database design you create can be implemented using either of two implementation languages provided by CA IDMS/DB:

- SQL DDL
- Non-SQL DDL

Design considerations are documented in this manual.

Note: For complete SQL DDL statements, see the *CA IDMS SQL Reference Guide*. For complete non-SQL DDL statements, see the *CA IDMS Database Administration Guide.*

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.

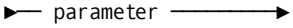
Indicates the end of a complete piece of syntax.

Indicates that the syntax continues on the next line.

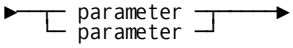
Indicates that the syntax continues on this line.

Indicates that the parameter continues on the next line.

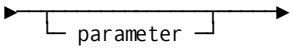
Indicates that a parameter continues on this line.



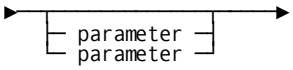
Indicates a required parameter.



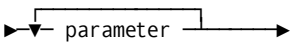
Indicates a choice of required parameters. You must select one.



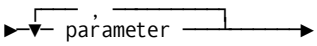
Indicates an optional parameter.



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



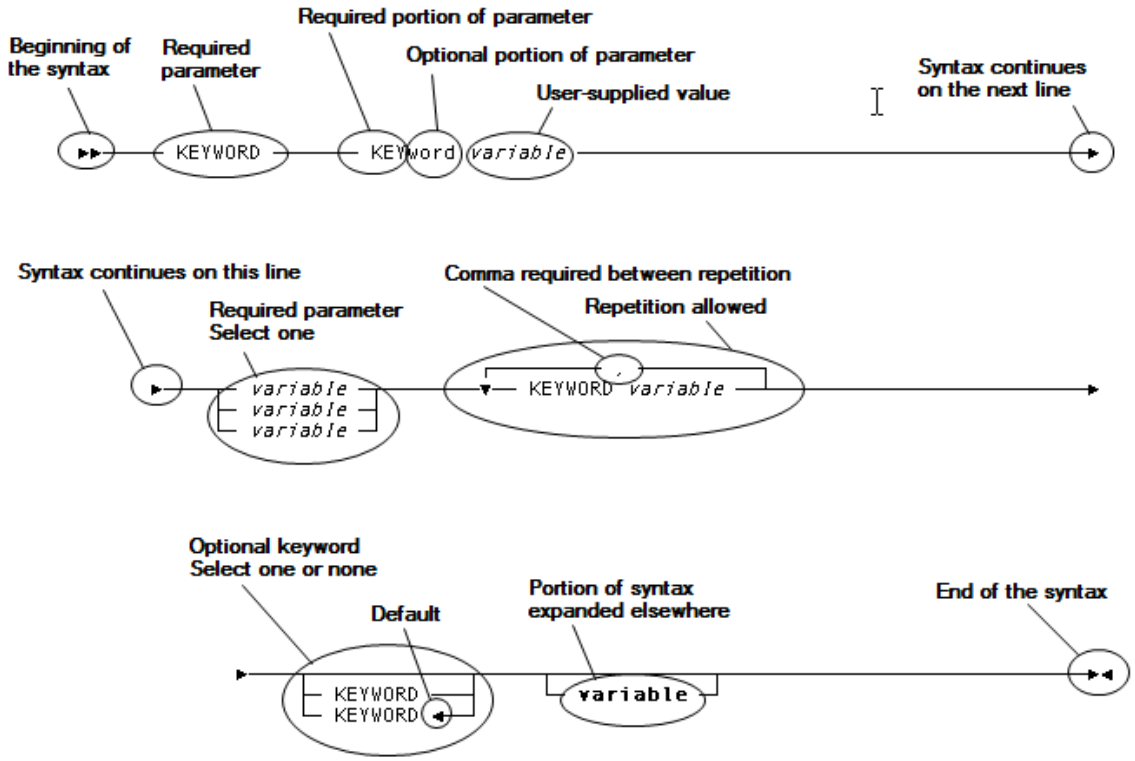
Indicates that you can repeat the parameter or specify more than one parameter.



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:



Chapter 2: Introduction to Logical Design

This section contains the following topics:

Overview

What is logical database design

Logical database design is the process of determining the logical data structures needed to support an organization's information resource. The logical design process helps you to implement a database that satisfies the requirements of your business organization.

Logical design is critical to the implementation of a corporate database. If your logical design is incomplete or has flaws, making changes to the means of data collection, storage, and protection can be costly later on. By using a well-conceived preliminary design, you can easily implement and test a database. A sound logical design therefore helps to ensure a successful implementation.

A complete and accurate logical design for a database helps to ensure:

- **Data independence**—The logical design process yields a database model that is independent of program or physical storage requirements. This model represents the way data structures appear to users. It does not specify how data structures are maintained in or processed by the computer.
- **Physical database flexibility**—Because the logical design is independent of storage and performance requirements, it can be used to implement a database used with any hardware or software system. During the physical design process, the logical design can be tailored to satisfy the needs of particular users or to suit a particular data processing environment.
- **Integrity**—The logical design identifies both the data maintained in your corporation and the rules of the business. These business rules can be used later to define integrity rules for the physical design.
- **User satisfaction**—The logical design represents data structures in a simple, understandable format. You can show the design to users at any stage of development without intimidating them. The logical design can be easily modified to incorporate users' suggestions and feedback.

There are many viable approaches available for logical database design. In this manual, we combine several design techniques, including systems analysis, the entity-relationship approach, and normalization.

Note: The entity-relationship approach was developed by Peter Chen. For further information on his approach to database design, see *Entity-Relationship Approach to Information Modeling and Analysis*, Peter P. Chen, editor, ER Institute (1981).

By using these techniques, you can create a logical model that consists of:

- Descriptions of the data required by each user application
- A comprehensive picture of the corporation's data

Determining the Users' Data Needs

Users of application programs require access to only selected portions of a database. Therefore, you need to develop a logical model that includes descriptions of the data required by each program.

Data tables

To the user of an application program, information in a CA IDMS/DB database will appear in the form of **data tables**. Data tables consist of columns and rows of related data. For example, a table might contain information about a company's departments, organized under headings such as DEPT ID, DEPT NAME, and DEPT BUDGET. A DEPARTMENT table with these categories of information is illustrated in the following diagram.

Information for company departments is maintained in the Department data table. A column represents a list of all department IDs. A row represents a single department.

Department data table

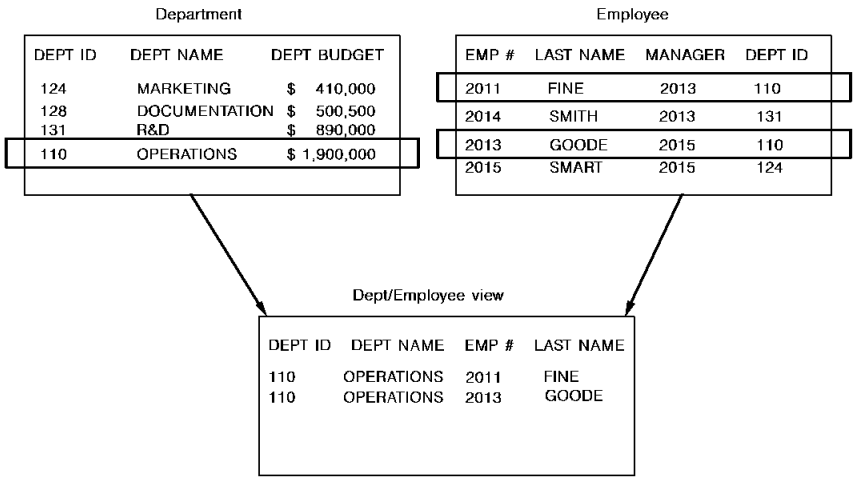
DEPT ID	DEPT NAME	DEPT BUDGET
124	MARKETING	$ 410,000
128	DOCUMENTATION	$ 500,500
131	RESEARCH AND DEVELOPMENT	$ 890,000
110	OPERATIONS	$ 1,900,000

Views

Users can manipulate columns and rows of data by accessing tables directly or by defining **views** of the database. Views enable users to select specified rows or columns or to combine information from two or more tables. For example, a view might use the relational join operation to combine information from the DEPARTMENT table and the EMPLOYEE table, as illustrated below.

Relational join operation

To show company employees with their departments, the DEPT/EMPLOYEE view uses the common DEPT ID column to join the Department and Employee data tables. This join operation selects all information from the tables that pertains to department 110. In the DEPT/EMPLOYEE view, the project operation has been used to include the DEPT ID and DEPT NAME columns from the DEPARTMENT table and the EMP # and LAST NAME columns from the EMPLOYEE table.


Department
DEPT ID
DEPT NAME
DEPT BUDGET
124
MARKETING
$ 410,000
128
DOCUMENTATION
$ 500,500
131
R&D
$ 890,000
110
OPERATIONS
$ 1,900,000
Employee
EMP #
LAST NAME
MANAGER
DEPT ID
2011
FINE
2013
110
2014
SMITH
2013
131
2013
GOODE
2015
110
2015
SMART
2015
124
Dept/Employee view
DEPT ID
DEPT NAME
EMP #
LAST NAME
110
OPERATIONS
2011
FINE
110
OPERATIONS
2013
GOODE

Determining the Corporation's Data Needs

As the database designer, you must understand *all* data used in your corporation. Once you have determined the user's information requirements, you need to develop a comprehensive picture of the corporation's data. Your logical design must include a complete description of this data.

Entity-relationship diagram

To represent the total picture, you can use the entity-relationship approach to logical design. With this approach, you develop an **entity-relationship diagram**, which serves as a model of the entire corporate enterprise. This diagram visually represents all data relationships that exist within the corporation.

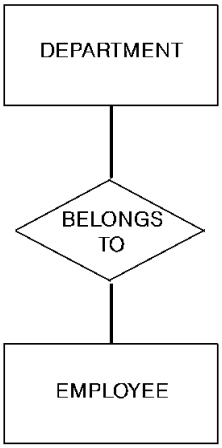
Entities

If data tables allow you to see the "trees" in a database, the arrangement of entities in an entity-relationship diagram helps you to represent the "forest." An **entity** is any general category of information used for business data processing. For example, the DEPARTMENT entity might describe information about the departments in a corporation, while the EMPLOYEE entity might describe company employees.

Entity-relationship diagramming

When two or more entities in a database share a relationship, their relationship can be graphically depicted on the entity-relationship diagram.

In the diagram below, the DEPARTMENT and EMPLOYEE data entities are related through the relationship BELONGS TO.



Overview of the Logical Design Process

During the initial stage of logical design, you identify the business problem that users hope to solve by creating a database. After interviewing several employees, you perform a thorough analysis of the business system, determining the processing functions performed by the organization and the flow of data during typical executions of these functions.

An analysis of the system provides documentation of the types of data required by users to perform their day-to-day business tasks. With this documentation, you can create the entity-relationship diagram.

Procedure

Logical database design involves the following procedures:

- Analyzing the business system
- Identifying the data entities (or data tables) and their relationships
- Identifying the data attributes
- Normalizing the data attributes
- Verifying that all business functions are supported by the logical design

Note: The first three procedures listed above are often performed concurrently. For example, in many instances, you will identify data entities, relationships, and attributes *as* you analyze the business system. By drawing a rough entity-relationship diagram during the systems analysis phase, you can sometimes simplify the design process.

Review the process

After you have performed the procedures listed above, you need to review the process and the available documentation with users in your corporation. As these users make suggestions for improvement, make appropriate changes to the design.

Each of the five major procedures of the logical design process is explained in detail in Chapters 3 through 7.

Chapter 3: Analyzing the Business System

This section contains the following topics:

Overview

Systems analysis is a necessary introduction to database design. Analyzing a corporate business system is a serious endeavor, about which many books have been written. It is not the purpose of this manual to describe the various methodologies available for performing systems analysis. Since this manual deals primarily with database design, it cannot present anything but an overview of systems analysis.

Analyzing the business system involves gathering information about the day-to-day functions of the organization, documenting this information, gathering more information, and so on, until a clear picture develops of the operations of the organization. To fully analyze the business system, you need to:

1. Define the general business functions.
2. Break down the general business functions into specific functions.
3. Identify the data elements used for functions and categorize them by subject.
4. Identify the business rules.
5. Review the results of analysis.

You can follow steps 1 through 5 below to perform a thorough analysis of your organization. Before you perform these procedures, you may need to write a description of the organization. This description will be used as the basis for systems analysis.

Organization description for the Commonweather Corporation

Below is a sample company description for the Commonweather Corporation.

```
Commonweather Corporation is a leader in the new, rapidly
expanding field of external climate control.  Commonweather
has offices in five locations.  Since its incorporation,
560 employees have been hired.  Most of these employees are
still with the company and have held, on the average, two
different positions.
```

Because Commonweather anticipates rapid growth, it has created an organizational structure that will be well suited to a company with many more employees. It has identified 41 different job titles and has created nine departments, each with its own department head. Several employees in each department have been appointed to supervisory positions and have hiring authority. Employees are, on occasion, assigned to head or participate in interdepartmental projects. In two years, the personnel department anticipates that there will be eight ongoing projects.

To facilitate the search for new employees, the personnel department has identified 68 skills that will need to be represented in the company's future employee base. When an employee is hired, the employee's level of expertise for each of these skills is identified.

The personnel department believes that by offering excellent employee benefits they can meet Commonweather's personnel needs. Therefore, they offer generous insurance benefits. Each employee is offered coverage in a life insurance plan, a dental plan, and a health plan (HMO or group-health). Employees can have complete family coverage or dependent coverage only.

A copy of each insurance claim filed by an employee for dental, hospital, or nonhospital services is sent to the personnel department. Each dental or nonhospital claim can be for up to ten dental or physician services. The personnel department submits all claims to the insurance companies. The department keeps a copy until the claim is paid; then the claim is thrown out. An employee cannot change coverage until all outstanding claims have been paid.

Step 1: Defining General Business Functions

What is a business function?

A business function is an activity performed during the day-to-day operations of an organization. The types of functions performed by a company determine the logical organization of the corporate database. To develop a complete logical design for a database, you therefore need to list all functions performed at your organization.

Often a business function can be broken down into several smaller functions. To avoid getting lost in the details, you should begin by listing the most general business functions.

Deriving the function list

By reviewing the company description, you can derive a list of the most general business functions. The following list of functions might be derived from the company description for the Commonweather Corporation:

- Hire employees
- Terminate employees
- Maintain employee information
- Maintain office and department information
- Maintain information about salaries and jobs
- Maintain skills inventory
- Maintain personnel information about projects
- Maintain information about employee insurance

Step 2: Defining Specific Business Functions

Smaller units of work

To break down the general business functions into smaller units of work, you need to think about what activities are involved in performing a particular business procedure.

For example, the general function *Maintain skills inventory* might involve these activities:

- Add a skill
- Add a skill for an employee
- Identify skills for an employee
- Identify skill level for an employee skill
- Identify all employees with a particular skill
- Identify all employees with a particular level of a particular skill
- Upgrade an employee skill level

Transactions

After you have broken down each general function into its component steps, you should be able to identify the most important application **transactions** of your organization. Your descriptions of these transactions can then be used by the MIS staff to develop application programs.

For further information on application development, see the *CA ADS Application Design Guide*.

In many instances, business functions can be broken down into *many* levels. Therefore, you may have to perform step 2 repeatedly to identify the most detailed functions of the business. For example, you might need to break down the function *Maintain skills inventory* several times before you can identify the application transactions.

Specific business functions for Commonweather Corporation

Below is a complete list of detailed business functions for the Commonweather Corporation.

1. Hire employees:

    a) Add an employee

    b) Assign an employee's position

    c) Assign an employee to an office

    d) Assign supervisory authority for an employee

    e) Assign supervisor for an employee

    f) Assign an employee to a department

2. Terminate employees:

    a) Delete an employee

    b) Delete an employee's position

    c) Remove an employee from an office

    d) Remove supervisory authority for an employee

    e) Remove an employee from a department

3. Maintain employee information:

    a) Assign or change an employee's position

    b) Assign an employee to or remove an employee from an office

    c) Assign an employee to or remove an employee from a department

    d) Assign or remove supervisory authority for an employee

    e) Assign or change supervisor for an employee

    f) List employees for a department

4. Maintain office and department information:

    a) Assign or delete an office

    b) Change an office address

    c) Add or delete a department

    d) Change a department head

5. Maintain information about salaries and jobs:

    a) Create a job

    b) Provide a job description

    c) Eliminate a job

    d) Establish job salaries

    e) Change job description or salary

6. Maintain skills inventory:

    a) Add a skill

    b) Add a skill for an employee

    c) Identify skills for an employee

    d) Identify skill level for an employee skill

    e) Identify all employees with a particular skill

    f) Identify all employees with a particular skill level

    g) Upgrade an employee skill level

7. Maintain personnel information about projects:

    a) Add a new project or delete a completed one

    b) Assign and remove employees from a project

    c) Assign or remove a project leader

    d) List names and phone numbers of all workers on a project

8. Maintain information about employee insurance:

   a) Add or remove a health insurance plan for an employee

   b) Identify the health insurance coverage for an employee

   c) Change coverage for an employee on a plan

   d) Add or change plan and coverage for an employee

   e) Add or delete a claim

   f) Show life and health insurance details for an employee

   g) Submit duplicate claim forms for an employee accident

Step 3: Listing the Data Elements

Identify data each function requires

After you have listed the business functions for your organization, you can begin to identify the data that each function requires. Your list of data elements (data table columns) will most likely expand and change as you gather more information about the organization. At this stage in the design process, simply list those elements that are clearly associated with each business task and group them according to general subject categories.

Consider using the following resources to identify data elements.

Interviews

Throughout the database design process, you conduct **interviews** with company personnel. Your meetings should give you an idea what data elements are required for particular business functions.

List of business functions

Many data elements can be identified in the **list of detailed business functions** (application transactions). Review your list of functions carefully to see if any elements can be recognized.

Data flow diagrams

To indicate the flow of information within the organization, you need to draw **data flow diagrams** (DFDs) for each of the general and specific business functions. A DFD should identify what information is needed to perform a particular function, where this information resides (logically, not in storage), and where it is likely to be moved during the course of processing. To identify the data flows, perform the following procedures:

1. Ask these questions:
    a. Where does the data come from?
    b. What happens to it when it reaches the system?
    c. Where does it go?
    d. What data should be restricted from user access?

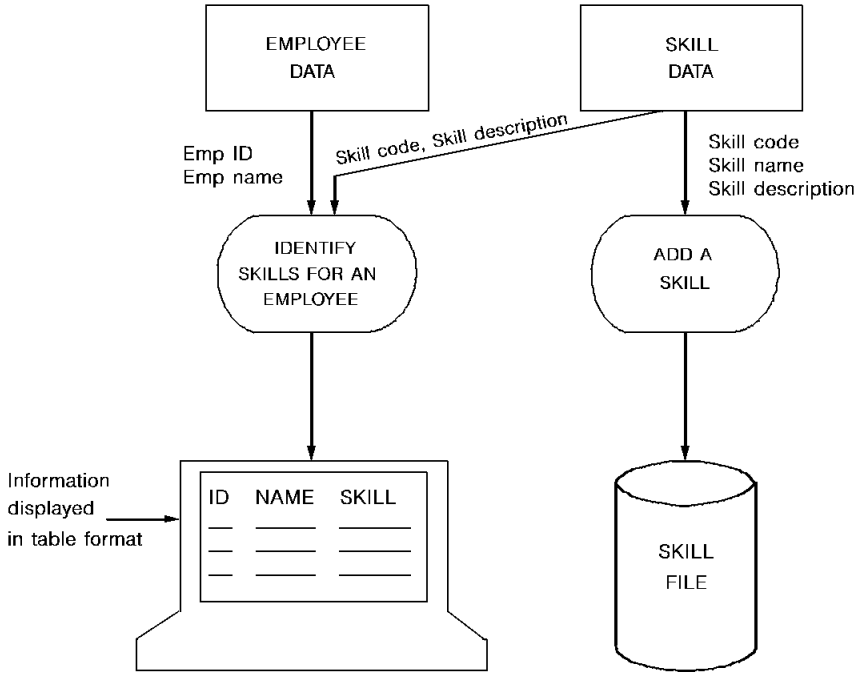
    Note: Once you have identified any restrictions that apply to the use of the information, you can begin to consider which security measures should be implemented for the system.
2. Identify the sources of information by defining the data stores:
    - People
    - Departments

- Documents

3. Verify the completeness of the information with users.

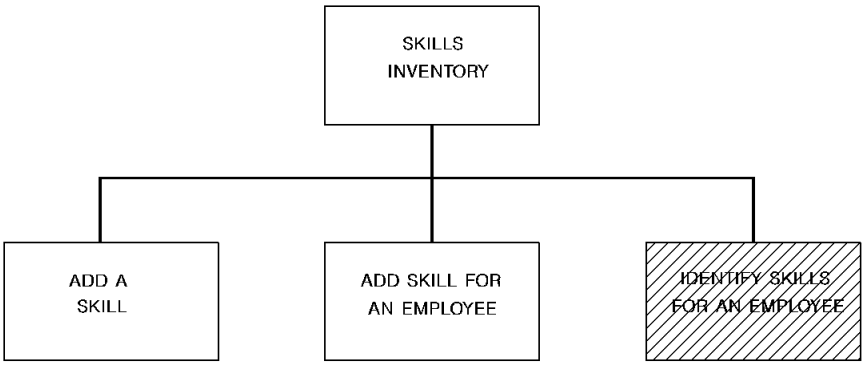
Data flow diagrams for a sample business function

The following diagram shows data flow diagrams (DFDs) for a general business function and its component steps.



Hierarchy plus Input-Process-Output diagrams

To indicate the flow of information within the organization, you may also want to draw **Hierarchy plus Input-Process-Output (HIPO) diagrams** for each of the business functions. A HIPO diagram can help you to identify what information is needed to perform a particular function. The diagram below shows a HIPO diagram for a sample business function.



HIPO overview diagram for "IDENTIFY SKILLS FOR AN EMPLOYEE" module

Input	Process	Output
• EMP ID	• RETRIEVE SKILL INFORMATION FOR THE EMPLOYEE	• EMP NAME • SKILL CODE • SKILL DESCRIPTION

Example

The following data elements might be accessed by the *Maintain skills inventory* function:

EMPLOYEE	SKILL
Employee name	Skill code
Employee ID	Skill name
Employee office	Date acquired
	Skill description

The grouping of elements under the categories EMPLOYEE and SKILL may change later on.

Step 4: Identifying the Business Rules

The rules of a business govern the execution of business functions against the database. Additionally, they define data integrity concerns that must be addressed during the course of database design. The business rules for your organization can be derived from the analysis of the company description, the function lists, the DFDs, and the HIPO diagrams. Compile a complete list of these rules.

Business rules for the Commonweather Corporation

The following is a list of business rules for the Commonweather Corporation.

1. There are currently five offices; expansion plans allow for a maximum of ten.
2. Employees can change position, department, or office.
3. There are 560 employees; allow for a maximum of 1000.
4. Records are maintained for an employee's previously held positions.
5. Each department has one department head and several members with supervisory positions with hiring authority.
6. Each office has a maximum of three telephone numbers.
7. When an employee is hired, his or her level of expertise in each of several skills is identified.
8. When an employee is hired, he or she automatically becomes a member of a particular department, and a particular office, and reports to a particular supervisor.
9. Each job description has several salary grades associated with it.
10. When hired, an employee is automatically covered by life insurance.

Step 5: Reviewing the Results of Analysis

Once you have performed steps 1 through 4 above, you need to review the materials you have gathered thus far. You need to ask yourself this question: *Has anything been overlooked?*

Making changes later on in the design process can sometimes be costly. Therefore, you should make sure that users have the chance to offer feedback at this point in the design process.

Documentation

By the time you have completed systems analysis, the following documentation should be available:

- General and specific function lists
- Data flow diagrams *or* HIPO diagrams for the functions
- List of data elements
- List of business rules

Using the dictionary

You can use the Integrated Data Dictionary (IDD) to document data elements and business rules.

Chapter 4: Identifying Entities and Relationships

This section contains the following topics:

Overview

By allowing you to document the total picture of an organization's data, the entity-relationship method of performing logical design allows you to:

- **Use a top-down approach for logical design**. To develop an entity-relationship diagram for a database, you define the most *general* categories of information first. Once you have identified these subject categories, you can then include more specific information in the design.
- **Demonstrate the semantic meaning of an organization's information**. This approach allows you to create a logical design for a database by analyzing descriptions of the organization that are written in everyday English. The entity-relationship diagram, the end product of logical design, accurately reflects the language used by employees to describe the organization. Therefore, this diagram can be reviewed and refined easily.

What are entities and relationships?

As you develop an entity-relationship diagram for a database, you identify each data **entity** and **relationship** used by the organization. An entity is a general category of business data that can be easily identified from the available documentation. A relationship defines a logical connection between two associated data entities. For example, the relationship REPORTS TO might identify a connection between a PERSON entity and a COMPANY entity.

Early in the logical design process, you need to determine the data entities and relationships necessary to fulfill the business functions of your organization. This chapter presents guidelines for identifying data entities and their relationships.

Identifying Data Entities

Identifying entities in the list of functions

Each data entity should appear as a *noun* in the list of sentences that define business functions, as illustrated below. Many nouns appear in the sentences that are *not* entities. Only nouns that describe data that is meaningful to the organization itself should be identified as entities.

Because each organization has unique data requirements, there is no single *correct* set of entities that can be derived from a list of functions. Given the same business functions, two organizations might select different key nouns, thereby creating unique lists of data entities.

1. Hire employees:
    a) Add an **employee**
    b) Assign an employee's **position**
    c) Assign an employee to an **office**
    d) Assign supervisory authority for an employee
    e) Assign supervisor for an employee
    f) Assign an employee to a **department**
2. Terminate employees:
    a) Delete an **employee**
    b) Delete an employee's **position**
    c) Remove an employee from an **office**
    d) Remove supervisory authority for an employee
    e) Remove an employee from a **department**

3. Maintain employee information:

   a) Assign or change an employee's **position**

   b) Assign an **employee** to or remove an employee from an **office**

   c) Assign an employee to or remove an employee from a department

   d) Assign or remove supervisory authority for an employee

   e) Assign or change supervisor for an employee

   f) List employees for a **department**

4. Maintain office and department information:

   a) Assign or delete an **office**

   b) Change an office address

   c) Add or delete a **department**

   d) Change a department head

5. Maintain information about salaries and jobs:

   a) Create a **job**

   b) Provide a job description

   c) Eliminate a job

   d) Establish job salaries.

   e) Change job description or salary.

6. Maintain skills inventory:

    a) Add a **skill**

    b) Add a skill for an **employee**

    c) Identify skills for an employee

    d) Identify skill level for an employee skill

    e) Identify all employees with a particular skill

    f) Identify all employees with a particular skill level

    g) Upgrade an employee skill level

7. Maintain personnel information about projects:

    a) Add a new **project** or delete a completed one

    b) Assign and remove employees from a project

    c) Assign or remove a project leader

    d) List names and phone numbers of all workers on a project

8. Maintain information about employee insurance:

    a) Add or remove a **health insurance plan** for an employee

    b) Identify the health insurance **coverage** for an employee

    c) Change coverage for an employee on a plan

    d) Add or change plan and coverage for an employee

    e) Add or delete a **claim**

    f) Show life and health insurance details for an employee

    g) Submit duplicate claim forms for an employee accident

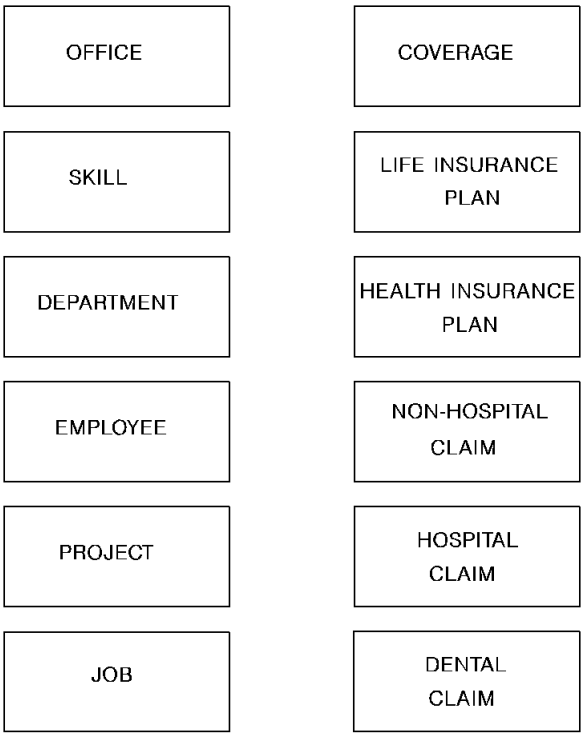
Steps to identify entities

To identify the data entities for your organization:

1. Determine which nouns in the list of business functions are the key nouns.
2. List these key nouns on a separate piece of paper.
3. Draw a rectangular box around each noun.

Data entities for the Commonweather Corporation

Below is a list of data entities that was derived from the list of functions for the Commonweather Corporation.

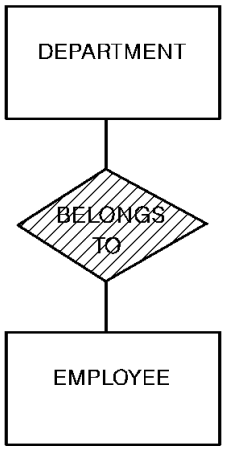


Identifying Relationships Among Entities

A relationship connects two associated data entities. The relationship between two entities can often be expressed with a *verb*. For example, the relationship between the DEPARTMENT entity and the EMPLOYEE entity might be expressed with the verb phrase BELONGS TO, since an employee belongs to a department in an organization.

Representing the relationship between two entities

The relationship between two entities is shown with a diamond. The name of the relationship is specified inside the diamond.



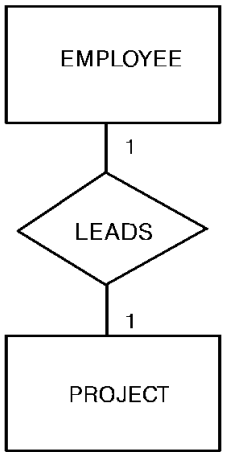
No hard-and-fast rule exists for determining data relationships for an organization. Data relationships depend on the requirements of the organization. The concept of *marriage*, for example, could be viewed as an entity type or a relationship between two people, depending on how the data was viewed.

Types of Data Relationships

Data entities in a database are related in one of three ways: one-to-one (1-1), one-to-many (1-M), and many-to-many (M-M). Each of these types of relationships is explained below.

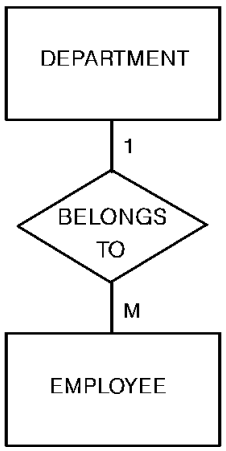
One-to-one (1-1)

In the one-to-one example below, for every EMPLOYEE entity occurrence in the database, there can exist only one corresponding PROJECT entity occurrence.



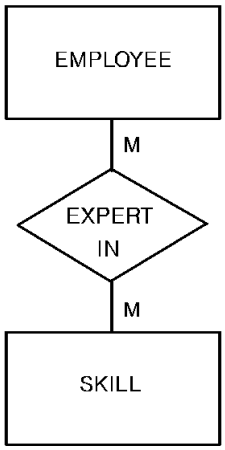
One-to-many (1-M)

In the one-to-many example below, for every DEPARTMENT entity occurrence in the database, there may exist one or more corresponding EMPLOYEE entity occurrences.



Many-to-many (M-M)

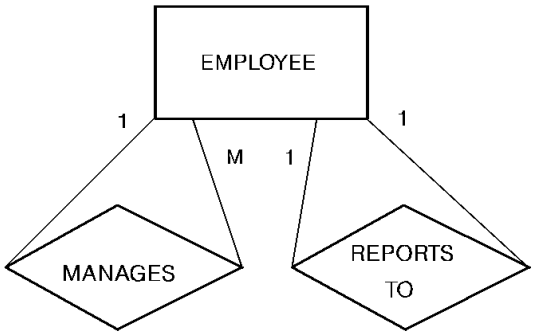
In the many-to-many example below, for every SKILL entity occurrence in the database, there can exist one or more corresponding EMPLOYEE entity occurrences; for every EMPLOYEE entity occurrence in the database, there can also exist one or more corresponding SKILL entity occurrences.



Other types of data relationships

In addition to relationships between two entity types, the following types of data relationships are acceptable in an entity-relationship model:

- **A relationship can be defined for only one entity type**. For example, to define a relationship between different employees in an organization, you might want to combine different data occurrences from the EMPLOYEE entity. In this case, the relationships among employees might be expressed as MANAGES and REPORTS TO, as shown in the entity-relationship diagram of Commonweather Corporation.



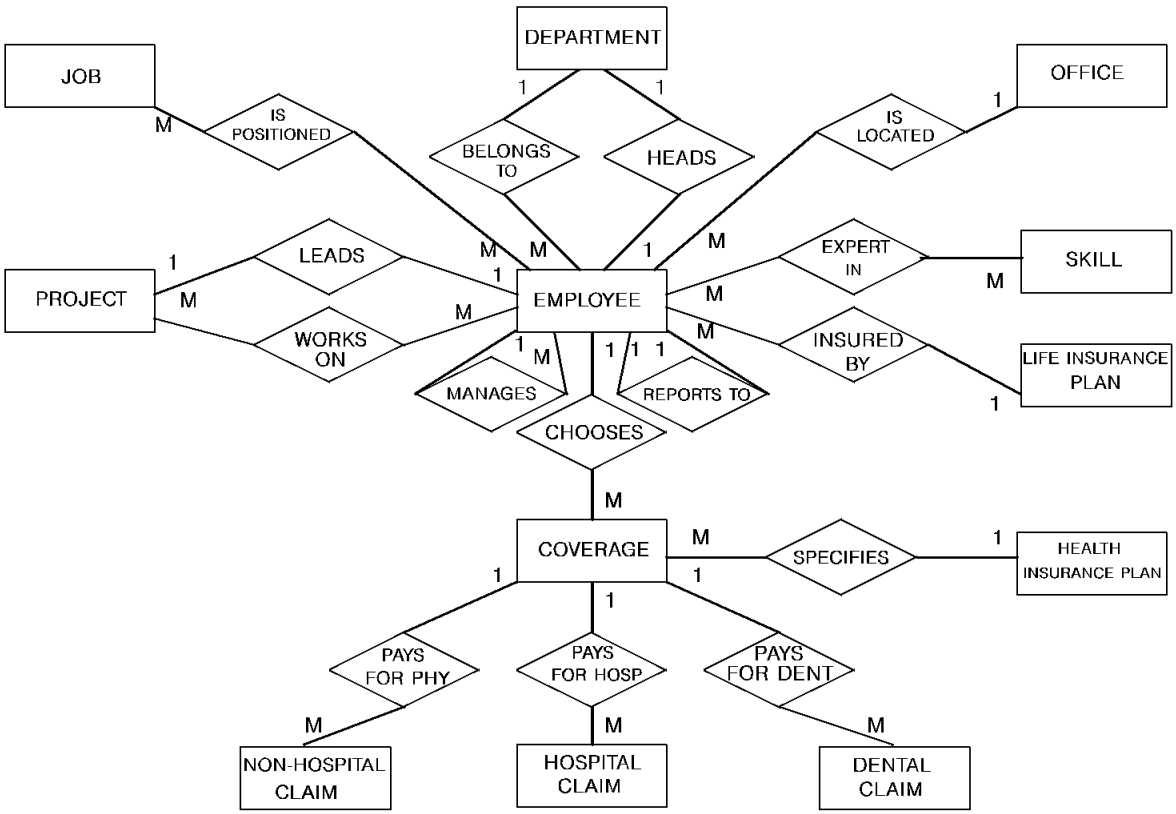
General Guidelines for Identifying Relationships

To identify the relationships between data entities, perform the following steps:

1. Using the list of business functions, identify relationships between entities as *verbs*. In those instances where no verb adequately expresses the relationship, join the two entity names to form a name for the relationship. For example, the DEPARTMENT and EMPLOYEE entities could be connected through the relationship BELONGS TO or through the relationship DEPT-EMPLOYEE.
2. List these key verbs between the entities they connect and draw a diamond around each one.
3. Associate entities with the appropriate relationships by connecting them with lines.
4. Label each relationship to show whether it is 1-1, 1-M, or M-M.

Entity-relationship diagram for Commonweather Corporation

The following diagram illustrates a simple entity-relationship diagram for the Commonweather Corporation.



Chapter 5: Identifying Attributes

This section contains the following topics:

Overview

Attributes and values

An **attribute** is the smallest unit of data that describes an entity or a relationship. A single occurrence of an attribute is called a **value**. For example, John Smith might be one of several values that exist for the attribute NAME of the entity EMPLOYEE. Several synonyms are used in the computer industry to refer to an attribute, including data item, data element, field, and column. All of these terms have roughly the same meaning.

In this chapter, we will identify the attributes that are associated with each entity and data relationship defined so far during the logical design process.

Identifying data attributes involves the following procedures:

1. Establishing naming conventions for the attributes
2. Identifying the attributes of each entity
3. Identifying the attributes for each relationship type

Each of these procedures is explained below.

Establishing Naming Conventions for the Attributes

Because the process of identifying attributes yields information from many different sources, the information can contain considerable redundancy. Users and data processing professionals have very different ways of perceiving the same data. The same piece of information might be called by several names, making it difficult to see that these names are **synonyms** for the same attribute. In addition, two different pieces of data might sometimes be called by the same name.

As soon as the business meaning of each attribute is clear, you should establish conventions for naming the attributes. Adopting a set of standardized naming conventions appropriate for the organization saves much time and confusion, and helps to ensure an efficient and effective design.

Identifying the Attributes of Each Entity

Each entity in a database is described by certain attributes. Attributes are those pieces of information about an entity that are required for processing performed by the business functions. By carefully examining the business functions, you can determine which attributes need to be maintained for each entity in the database.

Attribute categories

Attributes for a data entity fall into the following categories:

- **Unique keys**—To distinguish data occurrences, you need to identify unique keys. A unique key is an attribute or combination of attributes whose value or values uniquely identify an occurrence of an entity or relationship. Identification numbers and codes are typically used as unique keys, since their values are rarely modified.
- **Primary keys**—A primary key is a unique key that is used to represent an entity in a database. For example, the attribute EMP ID might be used as the primary key of the entity EMPLOYEE.
- **Secondary keys**—A secondary key is an attribute in a data entity that is used by certain business functions to access occurrences of that entity. For example, the EMP NAME attribute might be the secondary key for the entity EMPLOYEE.
- **Foreign keys**—A foreign key is an attribute of an entity or relationship that is also used as the primary key of another entity. A foreign key is used to relate two data entities. For example, to relate the DEPARTMENT and EMPLOYEE entities, you might define the DEPT ID attribute, which is the primary key of the DEPARTMENT entity, as the foreign key of the EMPLOYEE entity.

  By itself, a foreign key can never be the primary key of the entity in which it is stored. Since the DEPT ID attribute could never uniquely identify an occurrence of the EMPLOYEE entity, it could never be its primary key.

  However, a foreign key can be *part* of the primary key of an entity. In some instances, you need to combine a foreign key with another data element in an entity to create its primary key.
- **Non-key data**—All attributes of an entity that are not unique keys, primary keys, secondary keys, or foreign keys are considered non-key attributes. For example, the EMP ADDRESS attribute is a non-key attribute of the EMPLOYEE entity.

As you identify the attributes of each data entity, you need to determine whether the attributes are unique keys, primary keys, secondary keys, foreign keys, or non-key attributes.

Grouping the Attributes

You can identify the attributes associated with an entity by examining the following materials:

- List of business functions
- List of business rules
- List of data elements that you compiled during systems analysis

Attributes for entities

As you determine the attributes that are associated with a particular entity, you should list the attributes, as shown below:

OFFICE

- OFFICE CODE
- OFFICE ADDRESS
- OFFICE SPEED DIAL
- OFFICE AREA CODE
- OFFICE PHONE

DEPARTMENT

- DEPT ID
- DEPT NAME
- DEPT HEAD ID

SKILL

- SKILL CODE
- SKILL NAME
- SKILL DESCRIPTION

EMPLOYEE

- EMP ID
- EMP NAME
- SS NUMBER
- EMP ADDRESS
- EMP HOME PHONE
- DATE OF BIRTH
- DATE OF HIRE
- DATE OF TERMINATION
- STATUS

COVERAGE

- HEALTH PLAN CODE
- COVERAGE TYPE
- COVERAGE DESCRIPTION
- SELECTION DATE
- TERMINATION DATE

JOB

- JOB ID
- JOB TITLE
- JOB DESCRIPTION
- REQUIREMENTS
- MAX SALARY
- MIN SALARY
- NUMBER OF POSITIONS
- NUMBER OPEN
- SALARY GRADE

PROJECT

- PROJECT CODE
- PROJECT LEADER
- PROJECT DESCRIPTION
- EST START DATE
- ACT START DATE
- EST END DATE
- ACT END DATE

HEALTH INS PLAN

- HEALTH PLAN CODE
- INSCO NAME
- INSCO ADDRESS
- INSCO PHONE
- PLAN DESCRIPTION
- GROUP NUMBER

LIFE INS PLAN

- LIFE PLAN CODE
- INSCO NAME
- INSCO ADDRESS
- INSCO PHONE
- PLAN DESCRIPTION

DENTAL CLAIM

DENTAL CLAIM ID
EMP ID
COVERAGE TYPE
DATE OF CLAIM
PATIENT NAME
RELATION TO EMPLOYEE
PATIENT SEX
PATIENT DATE OF BIRTH
PATIENT ADDRESS
NUMBER OF DENTAL PROCEDURES
TOTAL CHARGES
DENTIST LICENSE NUMBER
DENTIST NAME
DENTIST ADDRESS
PROCEDURE ID
PROCEDURE DESCRIPTION
PROCEDURE FEE
SERVICE DATE

HOSPITAL CLAIM

HOSPITAL CLAIM ID
DATE OF CLAIM
EMP ID
COVERAGE TYPE
PATIENT NAME
RELATION TO EMPLOYEE
PATIENT SEX
PATIENT DATE OF BIRTH
PATIENT ADDRESS
DIAGNOSIS
TOTAL CHARGES
HOSPITAL NAME
HOSPITAL ADDRESS
ADMIT DATE
DISCHARGE DATE

NON-HOSPITAL CLAIM

NON-HOSPITAL CLAIM ID
DATE OF CLAIM
EMP ID
COVERAGE TYPE
PATIENT NAME
RELATION TO EMPLOYEE
PATIENT SEX
PATIENT DATE OF BIRTH
PATIENT ADDRESS
NUMBER OF PROCEDURES
TOTAL CHARGES
DIAGNOSIS
PHYSICIAN ID
PHYSICIAN NAME
PHYSICIAN ADDRESS
PROCEDURE ID
PROCEDURE DESCRIPTION
PROCEDURE FEE
SERVICE DATE

Identifying Unique Keys

An entity can have many attributes, but only some attributes uniquely identify occurrences of that entity. There might be more than one unique key in an entity. For example, the EMPLOYEE entity has two unique keys, EMP ID and EMP SS NUM.

For each entity, choose from among its attributes the ones that uniquely identify each occurrence. The attribute that best serves this purpose is a good candidate for a primary key. If there is no attribute that uniquely identifies an entity, it might be necessary to combine two or more attributes for a unique key or create an attribute that serves as a key.

Establishing Primary Keys

What is a primary key?

The **primary key** for each entity must be a unique key. From a business standpoint, the primary key should also be the most important element(s) in the entity. The requirements of your organization will determine which unique key attribute will be the primary key.

Suppose that you must select a primary key for the EMPLOYEE entity. Since both the EMP ID and EMP SS NUM attributes can be used to uniquely identify an occurrence of this entity, you need to select one of these keys. The EMP ID attribute is probably used most often for processing; therefore, this element is the best choice for the primary key.

Entities with primary keys

Once you have determined the primary key for an entity, you should mark this key with an asterisk (*), as shown below:

OFFICE

* OFFICE CODE
OFFICE ADDRESS
OFFICE SPEED DIAL
OFFICE AREA CODE
OFFICE PHONE

DEPARTMENT

* DEPT ID
DEPT NAME
DEPT HEAD ID

SKILL

* SKILL CODE
SKILL NAME
SKILL DESCRIPTION

JOB

* JOB ID
JOB TITLE
JOB DESCRIPTION
REQUIREMENTS
MAX SALARY
MIN SALARY
NUMBER OF POSITIONS
NUMBER OPEN
SALARY GRADE

PROJECT

* PROJECT CODE
PROJECT LEADER
PROJECT DESCRIPTION
EST START DATE
ACT START DATE

EMPLOYEE

* **EMP ID**
  EMP NAME
  SS NUMBER
  EMP ADDRESS
  EMP HOME PHONE
  DATE OF BIRTH
  DATE OF HIRE
  DATE OF TERMINATION
  STATUS

LIFE INS PLAN

* LIFE PLAN CODE
  INSCO NAME
  INSCO ADDRESS
  INSCO PHONE
  PLAN DESCRIPTION
  GROUP ID

DENTAL CLAIM

* DENTAL CLAIM ID
  EMP ID
  COVERAGE TYPE
  DATE OF CLAIM
  PATIENT NAME
  RELATION TO EMPLOYEE
  PATIENT SEX
  PATIENT DATE OF BIRTH
  PATIENT ADDRESS
  NUMBER OF DENTAL PROCEDURES
  TOTAL CHARGES
  DENTIST LICENSE NUMBER
  DENTIST NAME
  DENTIST ADDRESS
  PROCEDURE ID
  PROCEDURE DESCRIPTION
  PROCEDURE FEE
  SERVICE DATE

EST END DATE
ACT END DATE

HEALTH INS PLAN

* HEALTH PLAN CODE
  INSCO NAME
  INSCO ADDRESS
  INSCO PHONE
  PLAN DESCRIPTION
  GROUP ID

COVERAGE

* HEALTH PLAN CODE
* COVERAGE TYPE
  COVERAGE DESCRIPTION
  SELECTION DATE
  TERMINATION DATE

HOSPITAL CLAIM

* HOSPITAL CLAIM ID
  EMP ID
  COVERAGE TYPE
  DATE OF CLAIM
  PATIENT NAME
  RELATION TO EMPLOYEE
  PATIENT SEX
  PATIENT DATE OF BIRTH
  PATIENT ADDRESS
  DIAGNOSIS
  TOTAL CHARGES
  HOSPITAL NAME
  HOSPITAL ADDRESS
  ADMIT DATE
  DISCHARGE DATE

NON-HOSPITAL CLAIM

* NON-HOSPITAL CLAIM ID
EMP ID
COVERAGE TYPE
DATE OF CLAIM
PATIENT NAME
RELATION TO EMPLOYEE
PATIENT SEX
PATIENT DATE OF BIRTH
PATIENT ADDRESS
NUMBER OF PROCEDURES
TOTAL CHARGES
DIAGNOSIS
PHYSICIAN ID
PHYSICIAN NAME
PHYSICIAN ADDRESS
PROCEDURE ID
PROCEDURE DESCRIPTION
PROCEDURE FEE
SERVICE DATE

Identifying Weak Entities

What is a weak entity?

You may find that some entities in your database are identified only by their relationship with another entity. Such entities are called **weak entities**. Typically, a weak entity has a primary key that contains only one foreign key.

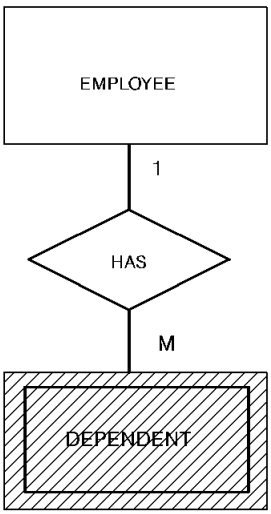
The entity DEPENDENT, for example, is a weak entity because it uses the primary key of the EMPLOYEE entity as part of its own primary key. Whenever an employee leaves the corporation, all information about that employee as well as any information about dependents must be erased from the database.

The attribute NAME is the only candidate for a primary key in the DEPENDENT entity, but NAME does not uniquely identify each occurrence of the DEPENDENT entity. Therefore, the primary key of the DEPENDENT entity must be a concatenation of the NAME attribute and the EMP ID attribute of the EMPLOYEE entity. This concatenated key provides the link between employees and their associated dependents.

Indicating a weak entity

You identify a weak entity on the entity-relationship diagram by drawing a double box around the entity, as shown in the diagram below.

DEPENDENT is a weak entity because it uses the primary key of the EMPLOYEE entity as part of its own primary key.



Identifying the Attributes for Each Relationship Type

Some data relationships have attributes that describe meaningful non-key information, others do not, as described below:

- **A one-to-one relationship sometimes carries non-key data**. An example of a one-to-one relationship is LEADS, where the business rules state that each project has a single leader, and one employee may be project leader for only one project. For this relationship, it may be important to carry the dates when the project leader begins and ends leadership responsibility.
- **A one-to-many relationship typically does not carry any non-key data**. The relationship LOCATES, for example, simply relates an employee to a particular office. There is no additional information about that relationship that is required by the business functions.
- **Many-to-many relationships usually *do* carry non-key information required by the business functions**. EXPERT IN, for example, carries information about a particular employee's level of expertise with a particular skill.
- **A self-referencing structure is a special kind of many-to-many relationship that sometimes carries non-key data**. For example, in the Commonweather Corporation, the relationship between workers and managers is defined as REPORTS TO and the relationship between managers and workers is defined as MANAGES. Non-key data about the REPORTS TO and MANAGES relationships might be the *dates* on which a relationship began and ended.

List the attributes

As you determine the attributes that are associated with a particular relationship, you should list these attributes, as follows:

IS LOCATED

* OFFICE CODE
* EMP ID

BELONGS TO

* DEPT ID
* EMP ID

LEADS

* PROJECT CODE
* WRKR EMP ID

HEADS

* DEPT ID
* HEAD EMP ID

WORKS ON

* PROJECT CODE
* WRKR EMP ID

IS POSITIONED

* EMP ID
* JOB ID
  SALARY
  COMMISSION PERCENT
  BONUS PERCENT
  OVERTIME RATE
  START DATE
  END DATE

EXPERT IN

* EMP ID
* SKILL CODE
  SKILL LEVEL
  DATE ACQUIRED

CHOOSES

* EMP ID
* HEALTH PLAN CODE
* COVERAGE TYPE

REPORTS TO

* WRKR EMP ID
* SUPR EMP ID
  WRKR BEGIN DATE
  WRKR END DATE

INSURED BY

* EMP ID
* LIFE PLAN CODE

MANAGES

* SUPR EMP ID
* WRKR EMP ID
  SUPR BEGIN DATE
  SUPR END DATE

PAYS FOR HOSP

* HEALTH PLAN CODE
* COVERAGE TYPE
* HOSPITAL CLAIM ID

SPECIFIES

* HEALTH PLAN CODE
* COVERAGE TYPE

PAYS FOR PHY

* HEALTH PLAN CODE
* COVERAGE TYPE
* NON-HOSPITAL CLAIM ID

PAYS FOR DENT

* HEALTH PLAN CODE
* COVERAGE TYPE
* DENTAL CLAIM ID

Identifying Attribute Characteristics

Attribute characteristics

At this time, you can identify characteristics of the attributes you have listed. Attribute characteristics include:

- Length
- Type (alphanumeric or numeric)
- Nullability

Null values

Sometimes you do not know the data associated with a particular attribute. The attribute might not be applicable to a particular entity occurrence, such as the phone number of an employee with no phone. Or the data simply might not be known yet, such as the credit rating of a new customer. Such attributes should allow null values. An attribute that does not allow null values requires that data always be entered.

Chapter 6: Normalizing the Data

This section contains the following topics:

Overview

Goals of normalization

You can use normalization techniques to refine the entity-relationship model. Once you have determined the entities, relationships, and attributes of a database, **you can use normalization procedures to ensure that each entity and relationship is designed in its simplest form**. The goal of normalization is to develop entities that consist of a primary key, together with a set of attributes whose values are determined solely by the value of the primary key.

In many instances, you will find that the entities you developed earlier are already organized in easy-to-use structures. The entity-relationship approach often breaks entities down into normalized structures naturally. In those instances when data entities and relationships are fully normalized, the normalization process does not result in any changes to the design.

Why Normalize Data?

Update anomalies

Through normalization, you can develop a database that is protected against **update anomalies**. Update anomalies are abnormal processing conditions that result from the execution of update functions against the database. Update anomalies sometimes compromise the integrity of the database; therefore, you need to design data entities and relationships that, when implemented as data tables, are fully protected against such anomalies.

Types of anomalies

The following examples illustrate two types of anomalies:

- **Deletion anomaly**—Suppose you want to delete some information from the following data table:

  JOB

EMP ID	JOB ID	SALARY GRADE	SALARY
1216	ADM	18	15000
1041	MGR	30	30000
1633	INST	23	22000
1063	ADM	18	18000

  In the JOB table, the SALARY GRADE depends only on the JOB ID. If you delete the row for employee 1041 in the JOB table, you therefore lose not only the fact that employee 1041 is a manager, but also the fact that the SALARY GRADE for a manager is 30.

- **Insertion anomaly**—Suppose you want to add some information to the JOB table. You want to enter the fact that a programmer has a SALARY GRADE of 21. Because of the structure of the JOB table, you cannot enter this information until someone actually has a job as a programmer.

Preventing anomalies

To prevent anomalies from occurring during deletions and insertions of rows in the JOB table, you might create two separate tables:

POSITION

EMP ID	JOB ID	SALARY
1216	ADM	15000
1041	MGR	30000
1633	INST	22000
1063	ADM	18000

JOB

JOB ID	SALARY GRADE
ADM	18
MGR	30
INST	23
PGMR	21

Now you can delete the row for employee 1041 in the POSITION table without losing the fact that the SALARY GRADE for a manager is 30. You can also specify that a programmer has a SALARY GRADE of 21 in the JOB table without first specifying a programmer's name.

By breaking down data tables into smaller tables, you prevent update anomalies from occurring.

Normal Forms of Data

All normalized data tables exist in one of the following normal forms:

- First normal form
- Second normal form
- Third normal form

A data table that exists in a particular normal form complies with the rules that define that form. A table that exists in second normal form satisfies the criteria for first normal form; in addition, a table in third normal form satisfies the criteria for both first and second normal forms.

Goal of normalization

Since the rules of third normal form are the most rigorous, they are also the most desirable. **The goal of the normalization process is to create data tables that are organized in third normal form**.

Note: Several database theorists have suggested that tables in third normal form can be broken down into even simpler structures. For example, some theorists recommend that tables be organized in fourth or fifth normal form. However, at the present time, it seems most practical to organize data tables in third normal form.

The first, second, and third normal forms of data organization are discussed in the following sections.

First Normal Form

A data table is in first normal form if each of the attributes of a given row contains a single value. A table in first normal form has no repeating groups.

Table not in first normal form

Since the following table contains a repeating element called BUDGET, it is *not* in first normal form:

Note: In these examples, primary key attributes are highlighted.

DEPARTMENT

DEPT ID	DEPT NAME	BUDGET
1000	OPERATIONS	50000
		30000
		40000
		30000

Table in first normal form

The following table is in first normal form:

DEPARTMENT

DEPT ID	DEPT NAME
1000	OPERATIONS
2046	DEVELOPMENT
3333	DOCUMENTATION
5653	MARKETING

Second Normal Form

A data table is in second normal form if it is in first normal form and its entire primary key determines the values of each of its attributes. When a table is in second normal form, each of the attributes is dependent on the whole key and not any part of the key.

Table in first normal form

The POSITION table shown below is in first normal form but *not* in second normal form:

POSITION

EMP ID	JOB ID	EMP NAME	SALARY GRADE	SALARY
1216	**ADM**	SMITH	18	15000
1041	**MGR**	JONES	30	30000
1633	**INST**	DAVIS	23	22000
1063	**ADM**	EVANS	18	18000

In the POSITION table shown above, the primary key is the concatenation of EMP ID and JOB ID. This table is not in second normal form because some of the non-key attributes are *dependent on a part of the primary key*. For example, the EMP NAME attribute is dependent on only EMP ID, while the SALARY GRADE attribute is dependent only on JOB ID.

Table in first and second normal forms

The following table is in both first and second normal forms:

POSITION

EMP ID	JOB ID	SALARY
1216	**ADM**	15000
1041	**MGR**	30000
1633	**INST**	22000
1063	**ADM**	18000

In the POSITION table shown above, the primary key is the concatenation of EMP ID and JOB ID. The POSITION table is in first normal form because it contains no repeating groups. It is in second normal form because the non-key attribute SALARY is dependent on the entire primary key (the concatenation of EMP ID and JOB ID). If a user knows an EMP ID value and a JOB ID value, the user can easily find out the SALARY for an employee who works in a particular job.

Third Normal Form

A data table is in third normal form if it is in second normal form *and* no non-key attribute determines the value of another non-key attribute; a table that is in third normal form contains no transitive dependencies among non-key attributes.

Table not in third normal form

The EMPLOYEE table shown below is *not* in third normal form:

EMPLOYEE

EMP ID	EMP NAME	DEPT ID	DEPT NAME
1216	SMITH	1000	OPERATIONS
1041	JONES	3500	MARKETING
1633	DAVIS	3400	DOCUMENTATION
1063	EVANS	2000	SUPPORT

Let's assume that EMP ID is the primary key of the EMPLOYEE table shown above. In this case, the table is not in third normal form because a non-key attribute has a transitive dependency on another non-key attribute. The DEPT NAME attribute is dependent on the DEPT ID attribute; a DEPT NAME value can be determined by the value of a particular DEPT ID.

Normalizing the table

To normalize the EMPLOYEE table shown above, you could break down this table into two separate tables:

EMPLOYEE

EMP ID	EMP NAME
1216	SMITH
1041	JONES
1633	DAVIS
1063	EVANS

DEPARTMENT

DEPT ID	DEPT NAME
1000	OPERATIONS
3500	MARKETING
3400	DOCUMENTATION
2000	SUPPORT

Since the EMP NAME attribute is not dependent on any other non-key attribute, the EMPLOYEE table shown above is in third normal form. In addition, since the DEPT NAME attribute is not dependent on any other non-key attribute, the DEPARTMENT table is also in third normal form.

Rules of first, second, and third normal forms

The following table summarizes the rules of each normal form of data organization.

Normal Form	Rules
First normal form	A data table is in first normal form if each of the attributes of a given row contains a single value; a table in first normal form has no repeating groups.
Second normal form	A data table is in second normal form if it is in first normal form and its entire primary key determines the values of each of its attributes. When a table is in second normal form, each of the attributes is dependent on the whole key and not any part of the key.
Third normal form	A data table is in third normal form if it is in second normal form and no non-key attribute determines the value of another non-key attribute. A table that is in third normal form contains no transitive dependencies among non-key attributes.

How To Normalize Data

The primary key for a data entity is used to determine whether the attributes for that entity satisfy the rules of second and third normal form. Sometimes you will need to organize the same list of attributes for an entity in different ways, depending on which attribute(s) is selected as the primary key.

Atomic primary key

The DENTAL CLAIM entity shown below is uniquely identified by an **atomic primary key**. An atomic primary key is a primary key that consists of a single attribute. The atomic primary key for the DENTAL CLAIM entity shown below is DENTAL CLAIM ID.

DENTAL CLAIM

* DENTAL CLAIM ID
EMP ID
COVERAGE TYPE
DATE OF CLAIM
PATIENT NAME
RELATION TO EMPLOYEE
PATIENT SEX
PATIENT DATE OF BIRTH
PATIENT ADDRESS
NUMBER OF DENTAL PROCEDURES
TOTAL CHARGES
DENTIST LICENSE NUMBER
DENTIST NAME
DENTIST ADDRESS
PROCEDURE ID
PROCEDURE DESCRIPTION
PROCEDURE FEE
SERVICE DATE

In all the examples that follow, primary key attributes are indicated with a star (*).

Listing Data in First Normal Form

After you have listed a particular entity and its attributes, you need to verify that the entity is in first normal form:

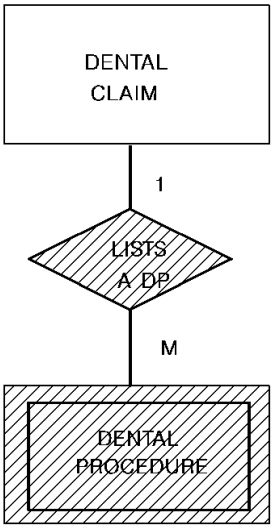
1. **Remove repeating groups**:
   a. For each repeating group identified, create a new entity.
   b. List its attributes.
   c. Identify its primary key.
   d. For each new entity created, create a relationship that relates it to the original entity in a 1-M manner.
2. **Update the E-R diagram** to reflect your changes.

Dental claim information in first normal form

The entities that describe dental claim information are listed in first normal form in the table below. The bold entity and relationship were added to organize the information in first normal form.

Data	Entity/ Relationship	Description
DENTAL CLAIM * DENTAL CLAIM ID EMP ID COVERAGE TYPE DATE OF CLAIM PATIENT NAME RELATION TO EMPLOYEE PATIENT SEX PATIENT DATE OF BIRTH PATIENT ADDRESS DENTIST LICENSE NUMBER DENTIST NAME DENTIST ADDRESS	Entity	Describes a dental claim for an employee.
LISTS A DP **(dental procedure)** *** DENTAL CLAIM ID** *** PROCEDURE ID**	Relationship	Relates DENTAL CLAIM to DENTAL PROCEDURE.

Data	Entity/ Relationship	Description
DENTAL PROCEDURE *** DENTAL CLAIM ID** *** PROCEDURE ID** **PROCEDURE DESCRIPTION** **PROCEDURE FEE** **SERVICE DATE**	Entity	Describes the procedures for a particular dental claim; this weak entity was derived from the DENTAL CLAIM entity because its attributes appeared as repeating elements.



Listing Data in Second Normal Form

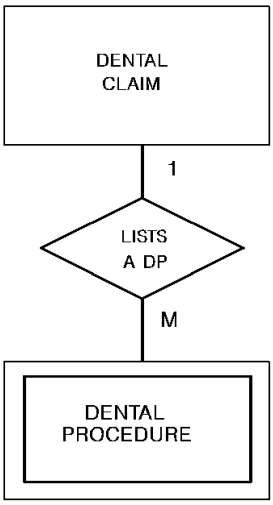
To verify that all entities are in second normal form, perform the following steps:

1. **Identify entities with compound keys.** Entities with compound keys are sometimes in first normal form but not in second normal form. Therefore, you need to carefully examine each entity that has more than one attribute in its primary key. By definition, entities with atomic keys are in second normal form (that is, if the entity contains no repeating groups and you selected an appropriate attribute as the primary key).
2. **Remove partially dependent attributes:**
   a. Locate any attributes that are dependent on only part of a compound key.
   b. Remove these attributes and create a new entity. Create a new relationship to relate the new entity to the entity from which it was removed.
3. **Update the E-R diagram to reflect these changes.**

Dental claim information in second normal form

The entities and relationships that describe dental claim information are listed in second normal form in the following table. No changes were made to organize the information in second normal form.

Data	Entity/ Relationship	Description
DENTAL CLAIM * DENTAL CLAIM ID EMP ID COVERAGE TYPE DATE OF CLAIM PATIENT NAME RELATION TO EMPLOYEE PATIENT SEX PATIENT DATE OF BIRTH PATIENT ADDRESS DENTIST LICENSE NUMBER DENTIST NAME DENTIST ADDRESS	Entity	Describes a dental claim for an employee.
LISTS A DP (dental procedure) * DENTAL CLAIM ID * PROCEDURE ID	Relationship	Relates DENTAL CLAIM to DENTAL PROCEDURE.
DENTAL PROCEDURE * DENTAL CLAIM ID * PROCEDURE ID PROCEDURE DESCRIPTION PROCEDURE FEE SERVICE DATE	Entity	Describes the procedures for a particular dental claim; this weak entity was derived from the DENTAL CLAIM entity because its attributes appeared as repeating elements.



Listing Data in Third Normal Form

To organize data entities in third normal form, perform the following steps:

1. **Remove transitively dependent attributes:**
   a. Locate any non-key attributes that are facts about another non-key attribute.
   b. Remove these attributes and create a new entity.
   c. Create a new relationship that relates the new entity to the original entity.
2. **Update the E-R diagram to reflect your changes.**

Dental claim information in third normal form

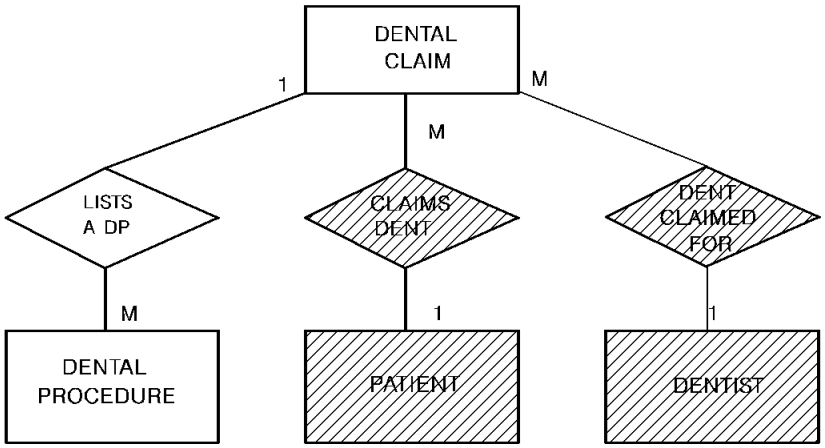
The entities and relationships that describe dental claim information are listed in third normal form in the following table.

The bold entities and relationships were added to organize the information in third normal form. Since none of the entities listed contain attributes that are dependent on part of the primary key, the information shown in this table is also in second normal form.

Data	Entity/ Relationship	Description
DENTAL CLAIM	Entity	Describes a dental claim for an employee.
* DENTAL CLAIM ID EMP ID DATE OF CLAIM		

Data	Entity/ Relationship	Description
LISTS A DP * DENTAL CLAIM ID * DENTAL PROCEDURE ID	Relationship	Relates DENTAL CLAIM to DENTAL PROCEDURE.
DENTAL PROCEDURE * DENTAL CLAIM ID * PROCEDURE ID PROCEDURE DESCRIPTION PROCEDURE FEE SERVICE DATE	Entity	Describes the procedures for a particular dental claim; this weak entity was derived from the DENTAL CLAIM entity because its attributes appeared as repeating elements.
CLAIMS DENT *** EMP ID** *** PATIENT NAME** *** DENTAL CLAIM ID**	Relationship	Relates PATIENT to DENTAL CLAIM.
PATIENT *** EMP ID** *** PATIENT NAME** **RELATION TO EMPLOYEE** **PATIENT DATE OF BIRTH** **PATIENT ADDRESS**	Entity	Describes a patient who makes a claim; this entity was derived from the DENTAL CLAIM entity to avoid transitive dependencies; in second normal form, the attributes RELATION TO EMPLOYEE, PATIENT DATE OF BIRTH, and PATIENT ADDRESS were dependent on the non-key attributes PATIENT NAME and EMP ID of DENTAL CLAIM.
DENT CLAIMED FOR *** DENTAL CLAIM ID** *** DENTIST LICENSE NUMBER**	Relationship	Relates DENTIST to DENTAL CLAIM.

Data	Entity/ Relationship	Description
DENTIST *** DENTIST LICENSE NUMBER** **DENTIST NAME** **DENTIST ADDRESS**	Entity	Describes the dentist who performs dental work for a patient; this entity was derived from the DENTAL CLAIM entity to avoid transitive dependencies; in second normal form, the attributes DENTIST NAME and DENTIST ADDRESS were transitively dependent on the non-key attributes DENTIST NAME and DENTIST ADDRESS of the DENTAL CLAIM entity.



Normalized Data for the Commonweather Corporation

The data entities and relationships for the Commonweather Corporation are listed in first, second, and third normal forms in the following tables.

Data entities for Commonweather in first normal form

The bold entities and relationships were added to organize the information in first normal form. Since none of the entities listed contain attributes that are dependent on part of the primary key, the information shown in this table is already in second normal form.

Data	Entity/ relationship	Description
OFFICE * OFFICE CODE OFFICE ADDRESS OFFICE SPEED DIAL OFFICE AREA CODE	Entity	Describes offices in which employees work.
CALLS *** OFFICE CODE** *** OFFICE PHONE**	Relationship	Relates OFFICE and PHONE.
PHONE *** OFFICE PHONE**	Entity	Describes office phones; this entity was derived from the OFFICE entity because its attributes appeared as repeating elements.
IS LOCATED * OFFICE CODE * EMP ID	Relationship	Relates EMPLOYEE and OFFICE.
SKILL * SKILL CODE SKILL NAME SKILL DESCRIPTION	Entity	Describes the skills for each employee.

Data	Entity/ relationship	Description
EXPERT IN * SKILL CODE * EMP ID SKILL LEVEL DATE ACQUIRED	Relationship	Relates SKILL and EMPLOYEE.
DEPARTMENT * DEPT ID DEPT NAME	Entity	Describes the departments that employees belong to.
BELONGS TO * DEPT ID * EMP ID	Relationship	Relates DEPARTMENT and EMPLOYEE.
HEADS * DEPT ID * EMP ID	Relationship	Relates DEPARTMENT and EMPLOYEE.
JOB * JOB ID JOB TITLE JOB DESCRIPTION REQUIREMENTS MAX SALARY MIN SALARY NUMBER OF POSITIONS	Entity	Describes the jobs employees perform within the company.
PAYS *** JOB ID** *** SALARY GRADE**	Relationship	Relates JOB and SALARY GRADE.

Data	Entity/ relationship	Description
SALARY GRADE *** JOB ID** *** SALARY GRADE** **GRADE MIN SALARY** **GRADE MAX SALARY**	Entity	Describes the salary grades for each job; this weak entity was derived from JOB because its attributes appeared as repeating elements.
IS POSITIONED * JOB ID * EMP ID SALARY OVERTIME RATE COMMISSION PERCENT BONUS PERCENT START DATE TERMINATION DATE	Relationship	Relates EMPLOYEE and JOB.
PROJECT * PROJECT CODE PROJECT DESCRIPTION EST START DATE ACT START DATE EST END DATE ACT END DATE	Entity	Describes projects that employees work on and lead.
WORKS ON * PROJECT CODE * EMP ID WO START DATE WO END DATE	Relationship	Relates EMPLOYEE and PROJECT.
LEADS * PROJECT CODE * EMP ID	Relationship	Relates EMPLOYEE and PROJECT

Data	Entity/ relationship	Description
REPORTS TO * WRKR EMP ID * SUPR EMP ID WRKR START DATE WRKR END DATE	Relationship	Relates those employees who are supervisors to other employees who are workers.
MANAGES * SUPR EMP ID * WRKR EMP ID SUPR START DATE SUPR END DATE	Relationship	Relates those employees who are workers to other employees who are supervisors.
EMPLOYEE * EMP ID EMP NAME SS NUMBER EMP ADDRESS EMP HOME PHONE DATE OF BIRTH DATE OF HIRE DATE OF TERMINATION STATUS	Entity	Describes company employees.
INSURED BY * EMP ID * LIFE PLAN CODE	Relationship	Relates EMPLOYEE and LIFE INS PLAN.
LIFE INS PLAN * LIFE PLAN CODE INSCO NAME INSCO ADDRESS INSCO PHONE PLAN DESCRIPTION GROUP NUMBER	Entity	Describes a life insurance plan for each employee.

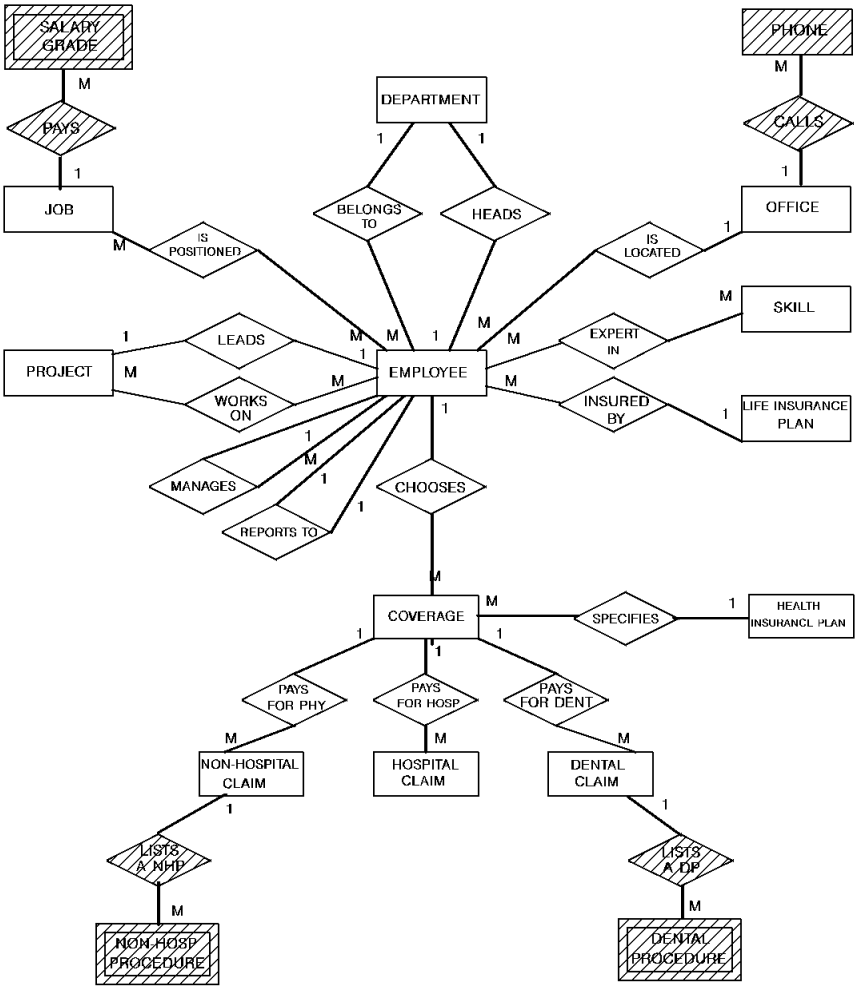
Data	Entity/ relationship	Description
CHOOSES * EMP ID * HEALTH PLAN CODE * COVERAGE TYPE	Relationship	Relates EMPLOYEE and COVERAGE.
COVERAGE * HEALTH PLAN CODE * COVERAGE TYPE COVERAGE DESCRIPTION SELECTION DATE TERMINATION DATE	Entity	Describes health coverage for each employee.
SPECIFIES * HEALTH PLAN CODE * COVERAGE TYPE	Relationship	Relates COVERAGE and HEALTH INS PLAN.
HEALTH INS PLAN * HEALTH PLAN CODE GROUP NUMBER INSCO NAME INSCO ADDRESS INSCO PHONE PLAN DESCRIPTION	Entity	Describes health insurance plans for employees.
PAYS FOR DENT * HEALTH PLAN CODE * COVERAGE TYPE * DENTAL CLAIM ID	Relationship	Relates COVERAGE and DENTAL CLAIM.

Data	Entity/ relationship	Description
DENTAL CLAIM * DENTAL CLAIM ID EMP ID COVERAGE TYPE DATE OF CLAIM PATIENT NAME RELATION TO EMPLOYEE PATIENT SEX PATIENT DATE OF BIRTH PATIENT ADDRESS DENTIST LICENSE NUMBER DENTIST NAME DENTIST ADDRESS	Entity	Describes a dental claim for an employee; in this example, the DENTAL CLAIM entity has an atomic key, DENTAL CLAIM ID.
LISTS A DP *** DENTAL CLAIM ID** *** PROCEDURE ID**	Relationship	Relates DENTAL CLAIM and DENTAL PROCEDURE.
DENTAL PROCEDURE *** DENTAL CLAIM ID** *** PROCEDURE ID** **PROCEDURE DESCRIPTION** **PROCEDURE FEE** **SERVICE DATE**	Entity	Describes the procedures for a particular dental claim; this entity was derived from the DENTAL CLAIM entity because its attributes appeared as repeating elements.
PAYS FOR HOSP * HOSPITAL CLAIM ID * HEALTH PLAN CODE * COVERAGE TYPE	Relationship	Relates COVERAGE and HOSPITAL CLAIM.

Data	Entity/ relationship	Description
HOSPITAL CLAIM	Entity	Describes a hospital claim for an employee.
* HOSPITAL CLAIM ID EMP ID COVERAGE TYPE DATE OF CLAIM PATIENT NAME RELATION TO EMPLOYEE PATIENT SEX PATIENT DATE OF BIRTH PATIENT ADDRESS DIAGNOSIS HOSPITAL NAME HOSPITAL ADDRESS HOSPITAL PHONE HOSPITAL CHARGES ADMIT DATE DISCHARGE DATE		
PAYS FOR PHY	Relationship	Relates COVERAGE and NON-HOSPITAL CLAIM.
* HEALTH PLAN CODE * COVERAGE TYPE * NON-HOSPITAL CLAIM ID		

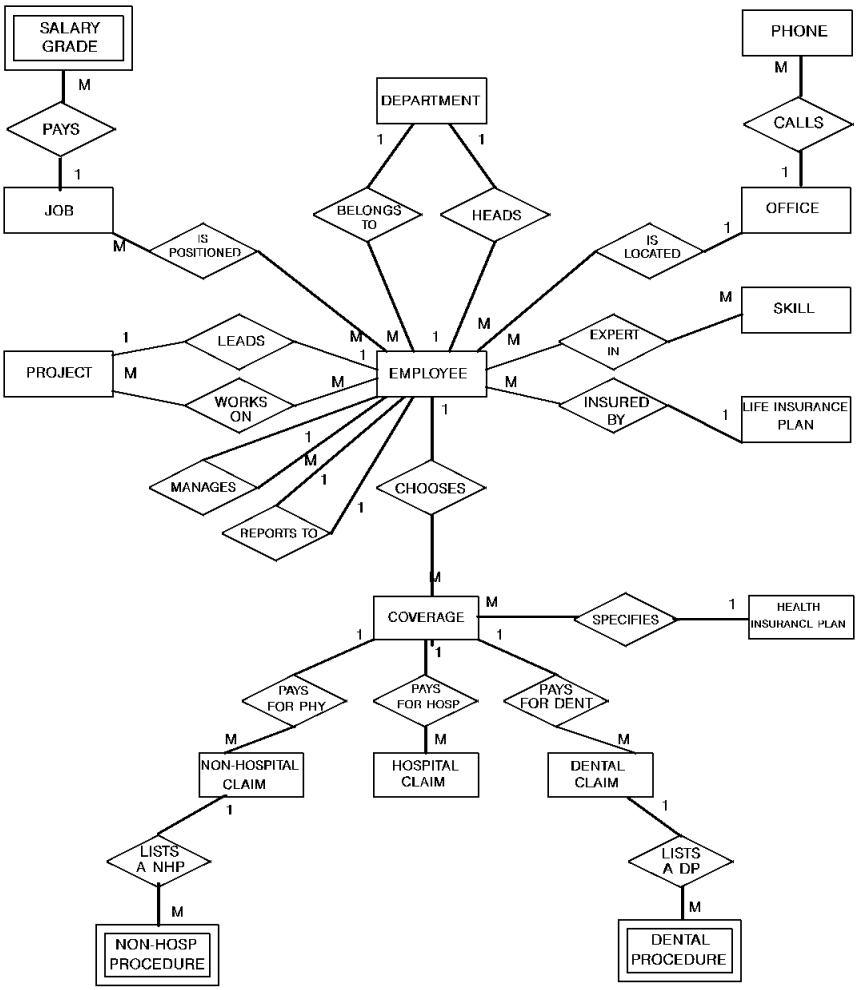
Data	Entity/ relationship	Description
NON-HOSPITAL CLAIM * NON-HOSPITAL CLAIM ID EMP ID COVERAGE TYPE DATE OF CLAIM PATIENT NAME RELATION TO EMPLOYEE PATIENT SEX PATIENT DATE OF BIRTH PATIENT ADDRESS DIAGNOSIS PHYSICIAN ID PHYSICIAN NAME PHYSICIAN ADDRESS NUMBER OF NON-HOSP PROCEDURES PHYSICIAN CHARGES	Entity	Describes a non-hospital claim for an employee.
LISTS A NHP *** NON-HOSPITAL CLAIM ID** *** NON-HOSPITAL PROCEDURE ID**	Relationship	Relates NON-HOSPITAL CLAIM and NON-HOSPITAL PROCEDURE.
NON-HOSPITAL PROCEDURE *** NON-HOSPITAL CLAIM ID** *** PROCEDURE ID** **PROCEDURE DESCRIPTION** **PROCEDURE FEE** **SERVICE DATE**	Entity	Describes the procedures for a particular hospital claim; this weak entity was derived from the NON-HOSPITAL CLAIM entity because its attributes appeared as repeating elements.

Data structure diagram showing Commonweather entities in first normal form



Data entities for Commonweather in second normal form

No changes were made to organize the information in second normal form.



Data entities for Commonweather in third normal form

The bold entities and relationships were added to organize the information in third normal form.

Data	Entity/ Relationship	Description
OFFICE	Entity	Describes the offices employees work in.
* OFFICE CODE		
OFFICE ADDRESS		
OFFICE SPEED DIAL		
OFFICE AREA CODE		

Data	Entity/ Relationship	Description
CALLS * OFFICE CODE * OFFICE PHONE	Relationship	Relates OFFICE and PHONE.
PHONE * OFFICE PHONE	Entity	Describes office phones; this entity was derived from the OFFICE entity because its attributes appeared as repeating elements.
IS LOCATED * OFFICE CODE * EMP ID	Relationship	Relates EMPLOYEE and OFFICE.
SKILL * SKILL CODE SKILL NAME SKILL DESCRIPTION	Entity	Describes skills for each employee.
EXPERT IN * SKILL CODE * EMP ID SKILL LEVEL DATE ACQUIRED	Relationship	Relates SKILL and EMPLOYEE.
DEPARTMENT * DEPT ID DEPT NAME	Entity	Describes departments in which employees work.
BELONGS TO * DEPT ID * EMP ID	Relationship	Relates DEPARTMENT and EMPLOYEE.
HEADS * DEPT ID * EMP ID	Relationship	Relates DEPARTMENT and EMPLOYEE.

Data	Entity/ Relationship	Description
JOB * JOB ID JOB TITLE JOB DESCRIPTION REQUIREMENTS MAX SALARY MIN SALARY NUMBER OF POSITIONS	Entity	Describes the jobs employees perform within the company.
PAYS * JOB ID * SALARY GRADE	Relationship	Relates JOB and SALARY GRADE.
SALARY GRADE * JOB ID * SALARY GRADE GRADE MIN SALARY GRADE MAX SALARY	Entity	Describes salary grades for each job; this entity was derived from the JOB entity because its attributes appeared as repeating elements.
IS POSITIONED * JOB ID * EMP ID SALARY OVERTIME RATE COMMISSION PERCENT BONUS PERCENT START DATE TERMINATION DATE	Relationship	Relates JOB and EMPLOYEE.

Data	Entity/ Relationship	Description
PROJECT * PROJECT CODE PROJECT DESCRIPTION EST START DATE ACT START DATE EST END DATE ACT END DATE	Entity	Describes the projects that employees work on.
WORKS ON * PROJECT CODE * EMP ID WO START DATE WO END DATE	Relationship	Relates EMPLOYEE and PROJECT.
LEADS * PROJECT CODE * EMP ID	Relationship	Relates EMPLOYEE and PROJECT.
REPORTS TO * WRKR EMP ID * SUPR EMP ID WRKR START DATE WRKR END DATE	Relationship	Relates those employees who are supervisors to other employees who are workers.
MANAGES * SUPR EMP ID * WRKR EMP ID SUPR START DATE SUPR END DATE	Relationship	Relates those employees who are workers to other employees who are supervisors.

Data	Entity/ Relationship	Description
EMPLOYEE * EMP ID EMP NAME SS NUMBER EMP ADDRESS EMP HOME PHONE DATE OF BIRTH DATE OF HIRE DATE OF TERMINATION STATUS	Entity	Describes company employees.
INSURED BY * EMP ID * LIFE PLAN CODE	Relationship	Relates EMPLOYEE and LIFE INS PLAN.
LIFE INS PLAN * LIFE PLAN CODE PLAN DESCRIPTION GROUP NUMBER	Entity	Describes the life insurance plan for each employee.
CHOOSES * EMP ID * HEALTH PLAN CODE * COVERAGE TYPE	Relationship	Relates EMPLOYEE and COVERAGE.
COVERAGE * HEALTH PLAN CODE * COVERAGE TYPE COVERAGE DESCRIPTION	Entity	Describes the health coverage chosen by each employee.
SPECIFIES * HEALTH PLAN CODE * COVERAGE TYPE	Relationship	Relates HEALTH INS PLAN and COVERAGE.

Data	Entity/ Relationship	Description
HEALTH INS PLAN * HEALTH PLAN CODE GROUP NUMBER PLAN DESCRIPTION	Entity	Describes the health insurance for each employee.
PROVIDES LIP *** LIFE PLAN CODE** *** INSCO NAME**	Relationship	Relates INS CO and LIFE INS PLAN.
PROVIDES HIP *** HEALTH PLAN CODE** *** INSCO NAME**	Relationship	Relates INS CO and HEALTH INS PLAN.
INS CO *** INSCO NAME** **INSCO ADDRESS** **INSCO PHONE**	Entity	Describes insurance companies; this entity was derived from the LIFE INS PLAN and HEALTH INS PLAN entities to avoid transitive dependencies; in second normal form, the attributes INSCO ADDRESS and INSCO PHONE were transitively dependent on the non-key attribute INSCO NAME.
PAYS FOR DENT * HEALTH PLAN CODE * COVERAGE TYPE * DENTAL CLAIM ID	Relationship	Relates COVERAGE and DENTAL CLAIM.
DENTAL CLAIM * DENTAL CLAIM ID DATE OF CLAIM	Entity	Describes a dental claim for an employee; in this example, the DENTAL CLAIM entity has an atomic key, DENTAL CLAIM ID.
LISTS A DP * DENTAL CLAIM ID * PROCEDURE ID	Relationship	Relates DENTAL CLAIM and DENTAL PROCEDURE.

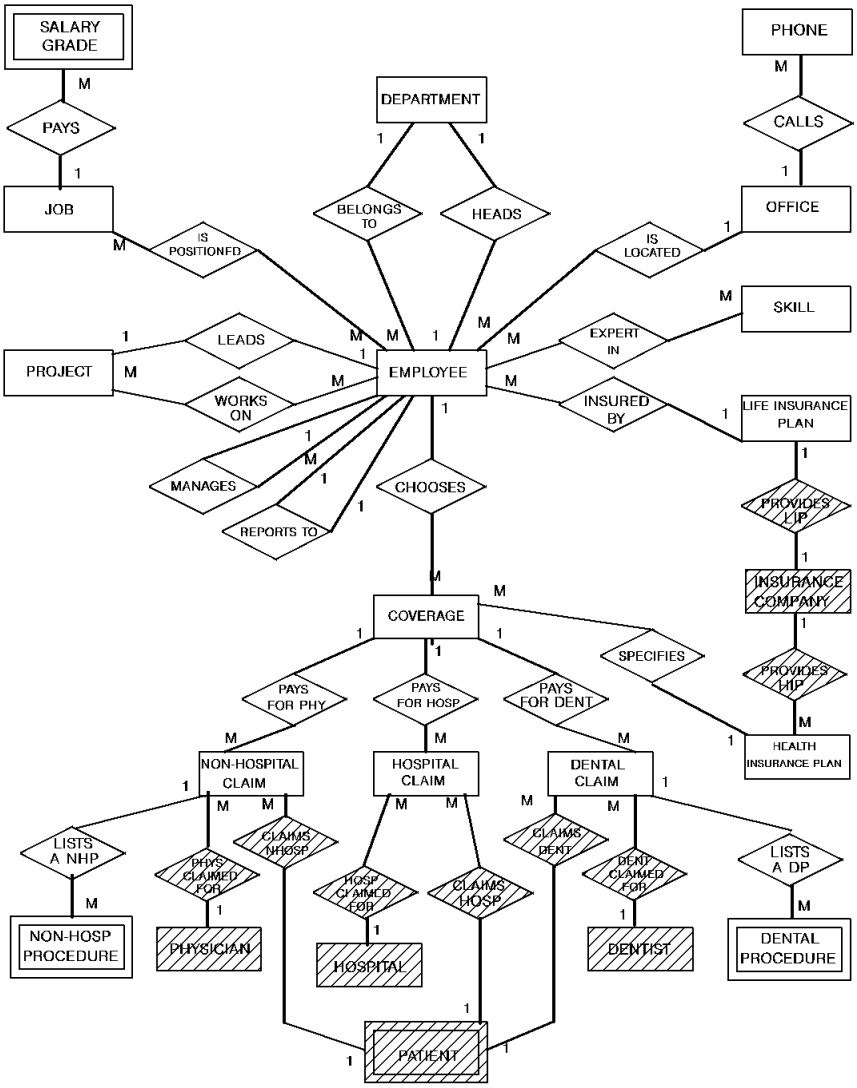
Data	Entity/ Relationship	Description
DENTAL PROCEDURE * DENTAL CLAIM ID * PROCEDURE ID PROCEDURE DESCRIPTION PROCEDURE FEE SERVICE DATE	Entity	Describes the procedures for a particular dental claim; this entity was derived from the DENTAL CLAIM entity because its attributes appeared as repeating elements.
DENT CLAIMED FOR *** DENTAL CLAIM ID** *** DENTIST LICENSE NUMBER**	Relationship	Relates DENTIST and DENTAL CLAIM.
DENTIST *** DENTIST LICENSE NUMBER** **DENTIST NAME** **DENTIST ADDRESS** **DENTIST PHONE**	Entity	Describes the dentist who performed dental work for a patient; this entity was derived from the DENTAL CLAIM entity to avoid transitive dependencies; in second normal form, the attributes DENTIST NAME and DENTIST ADDRESS were transitively dependent on the non-key attributes DENTIST NAME and DENTIST ADDRESS of the DENTAL CLAIM entity.
CLAIMS DENT *** DENTAL CLAIM ID** *** PATIENT NAME** *** EMP ID**	Relationship	Relates PATIENT and DENTAL CLAIM.
PAYS FOR HOSP * HOSPITAL CLAIM ID * HEALTH PLAN CODE * COVERAGE TYPE	Relationship	Relates COVERAGE and HOSPITAL CLAIM.

Data	Entity/ Relationship	Description
HOSPITAL CLAIM * HOSPITAL CLAIM ID EMP ID COVERAGE TYPE DATE OF CLAIM HOSPITAL CHARGES ADMIT DATE DISCHARGE DATE DIAGNOSIS	Entity	Describes a hospital claim for an employee.
HOSP CLAIMED FOR *** HOSPITAL CLAIM ID** *** HOSPITAL NAME**	Relationship	Relates HOSPITAL CLAIM and HOSPITAL.
HOSPITAL *** HOSPITAL NAME** **HOSPITAL ADDRESS** **HOSPITAL PHONE**	Entity	Describes the hospital in which a patient was treated; this entity was derived from the HOSPITAL CLAIM entity to avoid transitive dependencies; in second normal form, the attributes HOSPITAL ADDRESS and HOSPITAL PHONE were transitively dependent on the non-key attribute HOSPITAL NAME of the HOSPITAL CLAIM entity.
CLAIMS HOSP *** HOSPITAL CLAIM ID** *** PATIENT NAME** *** EMP ID**	Relationship	Relates PATIENT and HOSPITAL CLAIM.
PAYS FOR PHY * HEALTH PLAN CODE * COVERAGE TYPE * NON-HOSPITAL CLAIM ID	Relationship	Relates COVERAGE and NON-HOSPITAL CLAIM.

Data	Entity/ Relationship	Description
NON-HOSPITAL CLAIM * NON-HOSPITAL CLAIM ID DATE OF CLAIM DIAGNOSIS	Entity	Describes a non-hospital claim for an employee.
LISTS A NHP * NON-HOSPITAL CLAIM ID * PROCEDURE ID	Relationship	Relates NON-HOSPITAL CLAIM and NON-HOSPITAL PROCEDURE.
NON-HOSPITAL PROCEDURE * NON-HOSPITAL CLAIM ID * PROCEDURE ID PROCEDURE DESCRIPTION PROCEDURE FEE SERVICE DATE	Entity	Describes the procedures for a particular non-hospital claim; this entity was derived from the NON-HOSPITAL CLAIM entity because its attributes appeared as repeating elements.
PHYS CLAIMED FOR *** NON-HOSPITAL CLAIM ID** *** PHYSICIAN ID**	Relationship	Relates NON-HOSPITAL CLAIM and PHYSICIAN.
PHYSICIAN *** PHYSICIAN ID** **PHYSICIAN NAME** **PHYSICIAN ADDRESS** **PHYSICIAN PHONE**	Entity	Describes a physician who performed a service for a patient; this entity was derived from the NON-HOSPITAL CLAIM entity to avoid transitive dependencies; in second normal form, the attributes PHYSICIAN NAME, PHYSICIAN ADDRESS, and PHYSICIAN PHONE were transitively dependent on the non-key attribute PHYSICIAN ID of the NON-HOSPITAL CLAIM entity.
CLAIMS NHOSP *** NON-HOSPITAL CLAIM ID** *** PATIENT NAME** *** EMP ID**	Relationship	Relates NON-HOSPITAL CLAIM and PATIENT.

Data	Entity/ Relationship	Description
PATIENT *** EMP ID** *** PATIENT NAME** **RELATION TO EMPLOYEE** **PATIENT SEX** **PATIENT DATE OF BIRTH** **PATIENT ADDRESS**	Entity	Describes a patient who makes a claim; this entity was derived from the DENTAL CLAIM, HOSPITAL CLAIM, and NON-HOSPITAL CLAIM entities to avoid transitive dependencies; in second normal form, the attributes RELATION TO EMPLOYEE, PATIENT SEX, PATIENT DATE OF BIRTH, and PATIENT ADDRESS were transitively dependent on the non-key attributes PATIENT NAME and EMP ID of the DENTAL CLAIM entity; PATIENT is a weak entity related to EMPLOYEE.

Data structure diagram showing Commonweather entities in third normal form



Chapter 7: Validating the Logical Design

The final test of a logical design is whether it provides all the information needed for application processing. To verify that your logical database design is complete, you therefore need to simulate the flow of each business processing function through the database.

Tracing the access path

An **access path** shows the order in which data entities and their attributes are retrieved in the course of application processing. By tracing the access path of each general and specific business function, you can determine whether the database will support the processing needs of your organization. For clarity and readability, you need to draw a separate access path diagram for each business function.

Perform the following steps for each function:

1. **Identify the entry point for the function**. The entry point for a function is the first entity that it accesses in the database. You can determine the entry point for a function by analyzing the description of the function. (See Chapter 3, "Analyzing the Business System".) From the description of a particular function, you need to determine the most direct way to carry out the function.
2. **Identify all entities and relationships that must be accessed**. First make a list of all attributes required by the application. Then identify the entities and relationships that contain those attributes.

3. **Trace the direction of data flow**. To distinguish the direction of data flow from those lines that represent data relationships, you need to draw *dotted lines* to indicate the flow:

   a. Draw a dotted line from outside the diagram to the entry-point entity.

   b. Draw a dotted line through all entities and relationships that must be accessed. Do not be concerned about what keys might be necessary to move from one entity type to another. Retrieve an entity only if it has the attributes that you need to display or modify in some way.

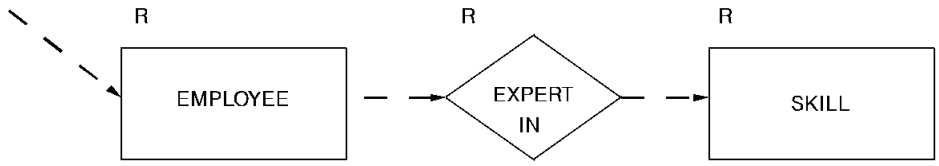
   c. Indicate the direction of data flow by drawing an arrow at the end of each dotted line.

4. **Determine the type of access**. Indicate on the access path diagram the type of access for each entity or relationship:

   - R— Read
   - C— Change
   - A— Add
   - D— Delete

Sample access path diagram

The following diagram illustrates a sample access path diagram for a general business function and its specific transactions.



As you trace the flow of each function, you may find that a particular application requires data that is not documented in the logical design. In the event that this happens, you need to make changes to the design to include this data. Once you have determined that the design contains all necessary data, you are prepared to develop a physical model for the database.

Chapter 8: Introduction to Physical Design

This section contains the following topics:

Overview

The database designer is responsible for efficient access to the database no matter how that database is implemented. This means that a complete logical and physical database design must take place prior to implementation.

In the first seven chapters, you worked through the process for creating a logical database design based on business functions and rules. You are now ready to make physical design decisions.

What is physical database design?

Physical database design is the process of tailoring the logical model to specific application performance requirements. During this phase of database design, you need to plan the best use of computer storage resources and provide for the most efficient data access.

At the conclusion of the logical design process, you should have documentation that represents the data model required to support the organization's information resource. As a result of normalization, you should also have an organized list of data entities. With these resources, you are prepared to make intelligent decisions about how to optimize database performance. This is the physical database design process.

Data Structure Diagram

The physical design process involves creating a diagram that serves as a model of the physical database. This diagram, known as a **data structure diagram**, visually represents the way data entities are related physically just as the entity-relationship diagram represents the way data entities are related logically. The data structure diagram also describes the storage characteristics of the data. Chapters 9 through 13 of this manual show you how to create a data structure diagram.

Steps in the Physical Database Design Process

The physical database design process involves creating a base physical design followed by refinements based on the implementation choice. The physical database design process involves the following steps:

1. Create a preliminary data structure diagram based on the logical database design.
2. Identify application performance requirements.
3. Assign location modes.
4. Evaluate and refine the physical database design.
5. Choose physical tuning options.
6. Minimize contention among transactions.

Physical Database Structures

Once you have created your design, you perform the necessary calculations to determine the amount of space required by your database, and then implement the database design using SQL or non-SQL data definition statements.

For further information on sizing the database, see Chapter 15, "Determining the Size of the Database". For further information on implementing the design, see the chapter Implementing Your Design.

No matter how you choose to define the database, certain physical database structures are used by CA IDMS/DB to implement your design.

For further information on physical database concepts, see the *CA IDMS Concepts and Facilities Guide* and *CA IDMS Database Administration Guide*.

Areas and pages

CA IDMS/DB subdivides the physical database into separate **areas**, each consisting of a set of contiguously numbered **pages**.

Areas are stored in operating system files, each page corresponding to one or more direct access blocks. CA IDMS/DB usually transfers an entire page of data in a single input/output operation.

While some database pages are reserved for space management, the majority of pages are used to hold user data in the form of **entity occurrences**. Each entity occurrence corresponds to a single row of an SQL-defined table or an instance of a record defined by a non-SQL schema.

A page can contain as many entity occurrences as space availability permits.

Segments

A segment defines the areas and files that contain the data in the database. A segment represents a physical database usually defined by a single schema. For the database to access the segment at runtime, the segment must be included in the definition of a **DMCL**.

DMCL

A DMCL is a collection of segment definitions that can be accessed in a single execution of CA IDMS/DB. The DMCL also specifies buffer characteristics, describes the buffer and files for journaling database activity, and identifies a database name table that the database uses at runtime to map a logical (or schema) definition of the database to specific segments.

A DMCL exists as a load module in a load (core-image) library and is used at runtime to determine where data required by an application is physically stored.

More Information

For more information on segments and the DMCL, see the *CA IDMS Database Administration Guide*.

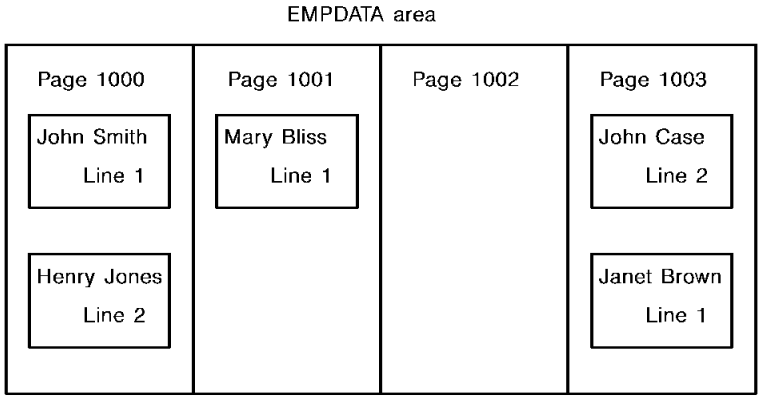
Database keys

CA IDMS/DB assigns a **database key** (db-key) to each record occurrence when it is entered into the database. The database key is the concatenation of the number of the page on which a record occurrence is stored and a line number. A line number is an index to an eight-byte structure called a line index. The line index is used to locate the record occurrence within the page. The database key uniquely identifies the record with which it is associated and never changes as long as the record remains in the database.

Structure of the physical database

The diagram below shows how areas, pages, and entity occurrences appear in the database.

The EMPDATA database area contains four pages and five entity occurrences. Each of the entity occurrences is uniquely identified by a database key. For example, the database key for the Mary Bliss occurrence is 1001:1.



SQL and Non-SQL Definitions

In CA IDMS, you have the choice of implementing your database design with either SQL or non-SQL definition statements. The choice of which definition language to use is based on the specific needs of your application.

Most of the physical design process is the same, regardless of which language is chosen. In those few areas of design implementation where the options differ for SQL and non-SQL, those options are clearly noted in this manual.

Likewise, there are some variances in the terminology used with each of the implementation languages. The accompanying table outlines sets of equivalent terminology.

Table of Terms

Logical/Physical Design Terminology	SQL Terminology	Non-SQL Terminology
Entity	Table	Record type
Entity occurrence	Row	Record occurrence
Data element	Column	Field/element
CALC location mode	CALC location mode	CALC location mode
Clustered location mode	Clustered location mode	VIA location mode
Parent	Referenced table	Owner
Child	Referencing table	Member
Relationship	Referential constraint	Set
Index	Index	Index

Chapter 9: Creating a Preliminary Data Structure Diagram

This section contains the following topics:

Developing a Data Structure Diagram

To derive a preliminary data structure diagram from an entity-relationship diagram, you need to:

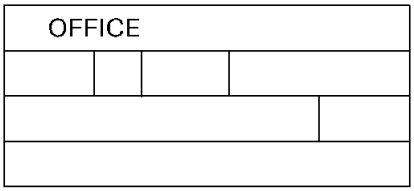
1. Represent entities.
2. Represent relationships between entities.
3. Estimate entity length (size of entities).

Follow the steps described below to create a preliminary data structure diagram for your database.

Representing Entities

Entities

Each entity in the logical database design is represented by an entity on the preliminary data structure diagram as shown below.



Each attribute identified during the logical database design process becomes a **data element** in the physical design. The names you used in the logical database design are also used in the physical design process.

Representing Relationships as Entities

Certain relationships defined during the logical design process should be represented as entities in the preliminary data structure diagram. These include:

- Relationships carrying non-key data
- Many-to-many relationships

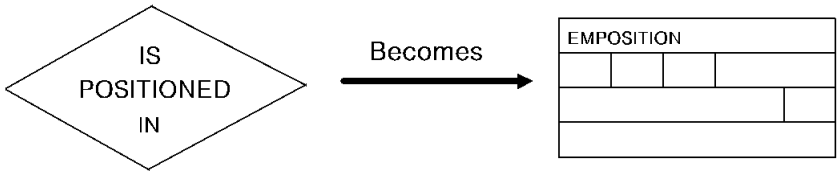
Another type of relationship, the self-referencing relationship, can become a separate entity in the preliminary data structure diagram or can carry the key to the relationship as a foreign key.

Each of these types of relationships is discussed below.

Relationships carrying non-key data

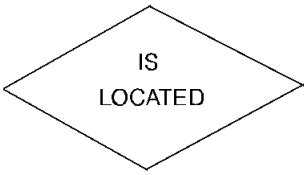
While most data relationships defined in the logical design contain only foreign keys, some carry both keys *and* non-key data. Relationships that contain non-key data must be represented as entities as you continue with the physical database design.

For example, because the relationship IS POSITIONED IN carries both keys and non-key data, it must be represented as an entity. Give this new entity an appropriate name.



Keys	Non-key data
JOB ID	SALARY
EMP ID	OVERTIME RATE
	COMMISSION PERCENT
	BONUS PERCENT
	START DATE
	TERMINATION DATE

However, the relationship IS LOCATED should *not* be represented as an entity because it contains only key information:



OFFICE CODE (**key**)
EMP ID (**key**)

Many-to-many relationships

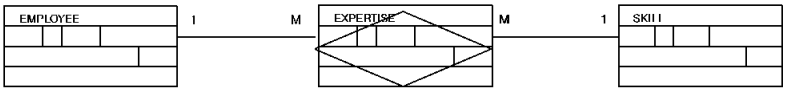
In a physical database design, you establish connections between related entities through **one-to-many** or **one-to-one** relationships. Each many-to-many relationship defined in the logical design must be converted to two one-to-many relationships. To make this change, you need to represent each many-to-many relationship as an entity, whether it contains non-key data or not. When you derive an entity from a many-to-many relationship, you create two one-to-many relationships, as shown below.

In the Commonweather Corporation, an employee can possess as many as five skills and a specific skill can be held by many employees. This situation establishes a many-to-many relationship between the SKILL and EMPLOYEE entities. Before you implement such a relationship under CA IDMS/DB, you must first create a new entity.

By replacing the many-to-many relationship between EMPLOYEE and SKILL with a new entity, you create two one-to-many relationships:

- A one-to-many relationship is created between EMPLOYEE and the entity EXPERTISE.
- Another one-to-many relationship is created between SKILL and EXPERTISE.

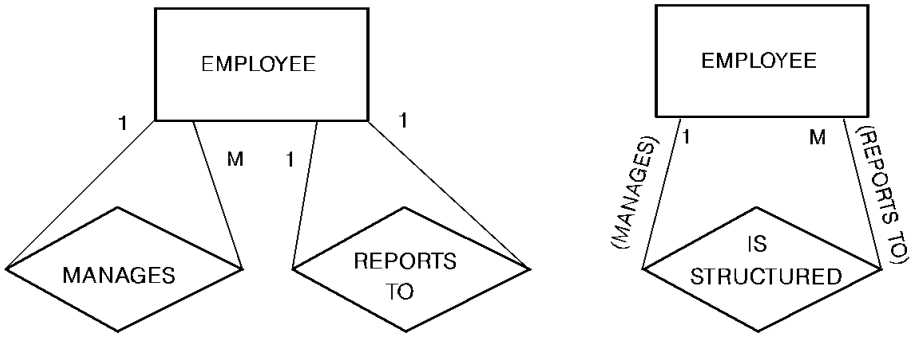
Name the new entities appropriately.



Self-referencing relationships

A **self-referencing relationship** allows users to combine information from different occurrences of the same entity. For example, to relate different employees in a company, an application program might combine data from different occurrences of the EMPLOYEE entity. A database user can then show employees and the managers they report to.

You may find more than one self-referencing relationship on a particular entity. If the relationships use the same keys, they are probably mirror images of each other. For example, MANAGES and REPORTS TO are two side of the same coin. Since they both use the same key and carry the same data, they are really one relationship.

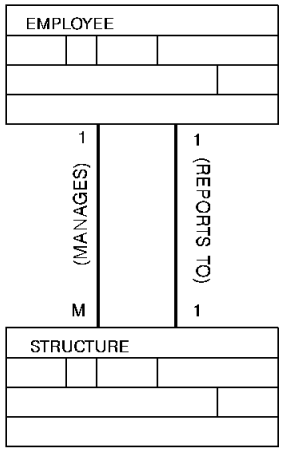


Replace the self-referencing relationship with an entity if any of the following are true:

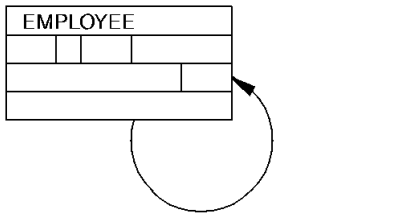
- If the self-referencing relationship carries data (for example, the date that the employee began to work for this manager)
- If you want to carry historical information (such as what managers an employee has had)
- If the self-referencing relationship is a many-to-many relationship

Replace the self-referencing relationship with an entity, specifying two relationships between the original entity and the new entity. These relationships can be one-to-many or one-to-one, depending on the logic behind them.

The following diagram shows how you might resolve a self-referencing relationship into an entity having two relationships with the primary entity: one one-to-many relationship and one one-to-one relationship. The new entity contains further information about the relationship between manager and employees.



If none of the above conditions apply, you can represent the relationship simply using a foreign key. In this case, the key of the manager would be carried as a foreign key in the EMPLOYEE entity. This approach will require fewer storage resources and therefore is recommended in those situations where it can be used.



(MANAGES-REPORTS TO)

Representing Relationships Between Entities

In the logical design process, you represented relationships between entities with diamonds and identified the keys associated with the relationship.

During the previous step ("Representing Entities") you changed each many-to-many relationship to two one-to-many relationships by creating a new entity. All relationships between entities should now fall into only two categories:

- One-to-many relationships
- One-to-one relationships

Representing the relationships

To represent the relationships in the preliminary data structure diagram, perform the following steps:

1. **For each relationship, draw a line** between the related entities.
2. **For each one-to-many relationship, place an arrow on the line** between the entities to identify the "many" side of the relationship.
3. **For each one-to-one relationship, do not draw an arrow** on the line between the entities.
4. **Name the relationship**. Usually the name is a concatenation of the two entities it relates.

   For example, the relationship between OFFICE and EMPLOYEE could be called OFFICE-EMPLOYEE and the relationship between SKILL and EXPERTISE could be called SKILL-EXPERTISE.
5. **Indicate the foreign key**.

   The foreign key will be shown as part of the definition of the relationship.

Foreign keys in a one-to-many relationship

In a one-to-many relationship, the key of the one entity is carried as a **foreign key** in the many entity.

For example, in the relationship between the entities OFFICE and EMPLOYEE, the key for the OFFICE entity (the one entity) is carried as a foreign key in the EMPLOYEE entity (the many entity).

Add the foreign key to the list of data elements associated with the appropriate entity and indicate each foreign key on the data structure diagram, as described below:

1. **Under the relationship name**, indicate the foreign key used in the relationship.

   For example, specify OFFICE CODE under the OFFICE-EMPLOYEE relationship to indicate that the data element OFFICE CODE is a foreign key for that relationship.

2. **Rename foreign keys used to establish self-referencing relationships**. Like any other entity that was originally a logical relationship, the entity used to define a self-referencing relationship carries as foreign keys the keys from each of the entities it relates. However, in this type of relationship, the two foreign keys must be derived from the same entity, EMPLOYEE.

   To avoid having two data elements with the same name (EMP ID) as keys to the entity, assign unique names to the foreign keys. For example, you might name the keys MGR ID and EMP ID to distinguish managers from workers.

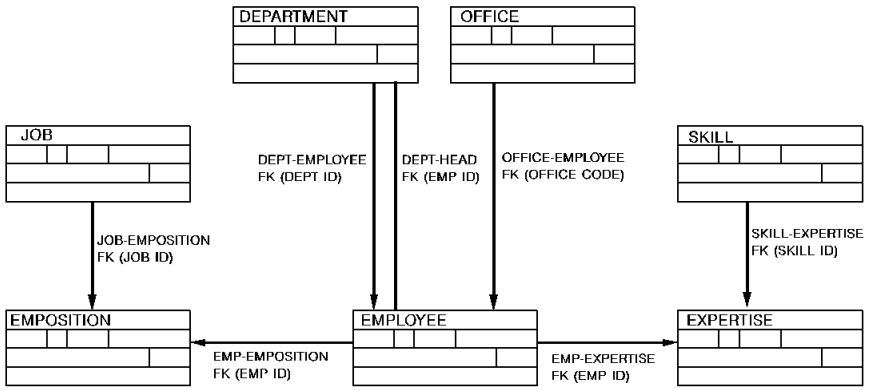
Note: The foreign key in a self-referencing relationship must be nullable. If it were not nullable, the first piece of data stored could not satisfy the referential integrity of the relationship. For example, the first employee stored would carry a manager ID that would not match an existing employee ID, as the integrity of the relationship requires. If the self-referencing relationship carries data, that data must also be nullable.

Foreign keys in a one-to-one relationship

In a one-to-one relationship, the foreign key can be placed in either entity participating in the relationship. Usually, you can conserve space by placing the foreign key in one of the two entities. For example, if there is a relationship between DEPARTMENT and EMPLOYEE to indicate which employee is head of a department, you can conserve space by placing the EMP ID of the head of the department in the DEPARTMENT entity rather than the other way around since there will typically be far more employees than departments.

Diagramming relationships between entities

The diagram below shows a portion of the data structure diagram for Commonweather after your changes have been made.



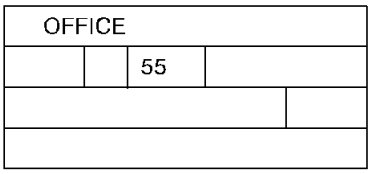
Estimating Entity Lengths

Once the entity types have been identified, you should estimate the length of each entity. To calculate each entity's length, add up the length of the data elements contained in the entity. Don't forget to include foreign keys residing in that entity. If the entity has a variable length, estimate the maximum possible length of the entity.

Although the lengths of entities may change as you refine the physical design, it is useful to have an estimate of the size of an entity during the design process.

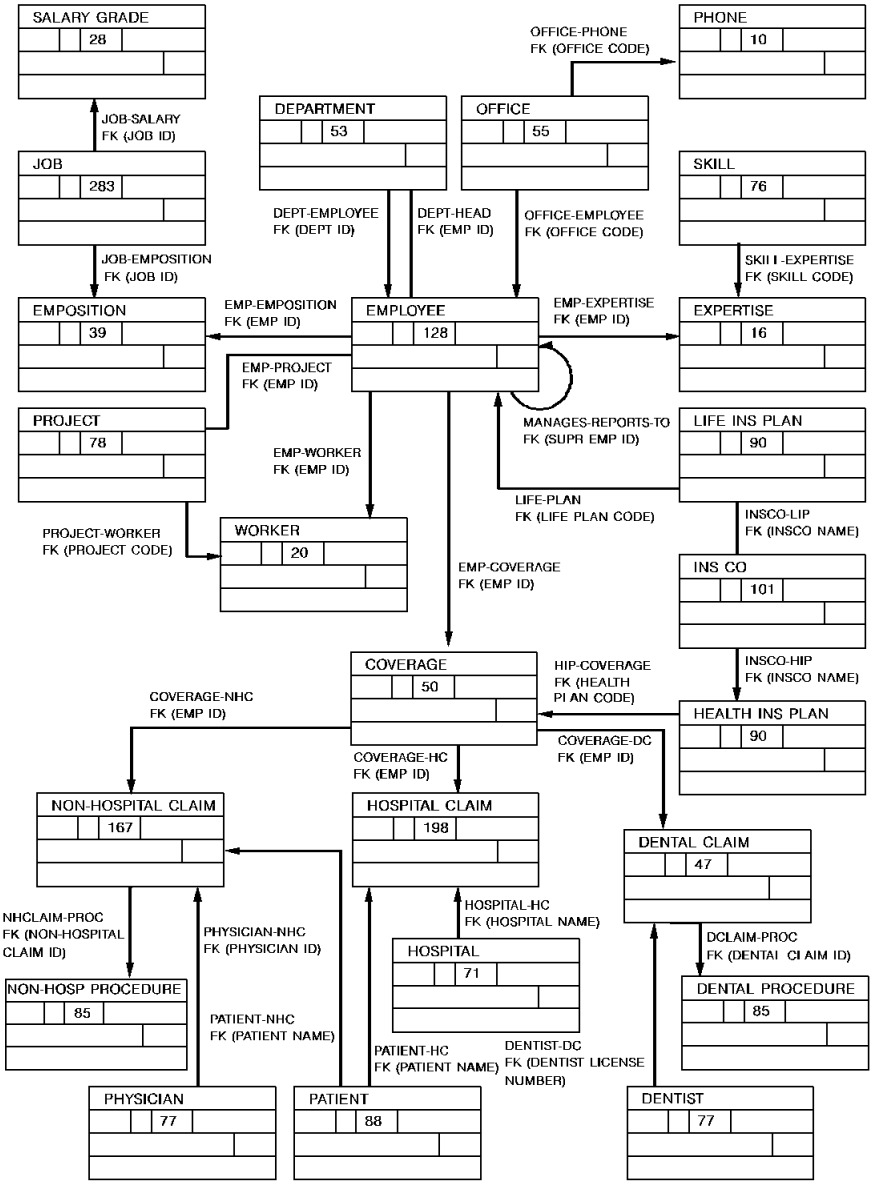
Indicating the length

Once you have determined the length of a particular database entity, you can indicate this information in the data structure diagram. The example below shows the OFFICE entity with a length of 55.



Preliminary Data Structure Diagram for Commonweather Corporation

Below is the preliminary data structure diagram for Commonweather Corporation. It represents entities, relationships, foreign keys, and estimated entity lengths.



Chapter 10: Identifying Application Performance Requirements

This section contains the following topics:

Overview

After creating the preliminary data structure diagram, you need to interview company employees who can help you determine the application requirements for the database so that you can refine that database structure.

Performance and storage requirements

As you gather information from users, you need to identify both the performance and storage requirements of the system:

- Establish performance requirements for transactions.
- Prioritize transactions.
- Determine how often each transaction will be executed.
- Identify access requirements for each transaction.
- Determine the database entry point and access key for each transaction.
- Project growth patterns.
- Determine the number of entity occurrences in each relationship.
- Determine how often each database entity will be accessed.

The requirements of the system determine how you should design the physical database model. For example, the requirements of a particular application can help you to define the page size for a database area.

Making design decisions

You will use the information that you gather at this stage in the physical design process to make several design decisions later on, as shown below.

Information gathered in this chapter	Used in...
■ Performance requirements for transactions ■ Transaction priorities ■ Access requirements ■ Database entry points and access keys	Refining the Physical Design (Chapter 12)
■ How often each transaction will be executed ■ How often each entity will be accessed	Minimizing Contention Among Transactions (Chapter 14)

Information gathered in this chapter	Used in...
■ Projected growth patterns ■ Number of entity occurrences in each relationship	Determining the Size of the Database (Chapter 15)

Establishing Performance Requirements for Transactions

Employees depend on fast computer turnaround to accomplish their day-to-day work. To ensure satisfactory turnaround time, you should establish performance requirements for the system.

Since company personnel have varying information requirements, you need to define separate performance requirements for each transaction. While some transactions perform high-volume, routine processing, such as payroll, inventory, and budgets, others enable end users to make ad hoc requests for information.

Company personnel measure the efficiency of a transaction by the amount of work it can perform and the amount of time it requires to perform the work. If you help employees to define realistic expectations of transaction performance, you can set performance requirements for the system that will be acceptable to the user community.

Processing modes

For each transaction, select a **mode** of computer processing that meets the needs of users without degrading system performance. For example, you might decide to execute a high-volume processing task as a batch job, while allowing end users to make ad hoc requests for data through an online application.

Once the processing mode has been established, define appropriate **performance requirements** for the transaction. Your requirements will vary depending on the mode of processing: while a 12-hour turnaround time might be acceptable for a large batch program, a 5-minute response time will be unsatisfactory for an online application.

Sample Transactions

The following table shows performance requirements for three sample transactions at the Commonweather Corporation.

Transaction	Processing Mode	Time
Add or delete a claim	Online	3 seconds
List of employees for an office	Batch	15 minutes
Show salary grade for all jobs	Online	6 seconds

Considerations

Your requirements should take into consideration the resources available with the computer system. If the resources are not adequate to meet the established performance requirements, you will need to modify the expectations of the user community or acquire additional resources.

Prioritizing Transactions

Every data processing department must prioritize requests for transactions. For example, when a high-level executive requires access to vital organization information, the data processing department tries to provide this information immediately.

As the DBA, you are responsible for ensuring that critical transactions execute in an efficient manner. To optimize performance, you need to schedule data processing tasks according to specific organization priorities.

Assigning priorities to transactions

The following table shows how you might prioritize three typical transactions.

Establish a HIGH priority for transactions that are vital to the operations of the organization. For example, you might specify a HIGH priority for a transaction that services the information needs of upper-level managers in the organization.

Sample Transactions

Transaction	Processing Mode	Time	Priority
Add or delete a claim	Online	3 seconds	High
List of employees for an office	Batch	15 minutes	Medium
Show salary grade for all jobs	Online	6 seconds	Low

Determining How Often Transactions Will Be Executed

Early in the design process, you need to determine how often each transaction will be executed. This can give you an indication of how the transaction might affect the overall performance of the system.

To determine how often particular transactions will be executed:

- Find out the hours when each transaction will be run.
- Create a preliminary schedule of batch update and reporting program runs.
- Once you have created a schedule of processing jobs, estimate how often each transaction will be executed during the hours when it is typically run.

Sample transactions

The following table shows how often three typical transactions will be executed.

Transaction	Processing Mode	Time	Priority	Frequency of Access
Add or delete a claim	Online	3 seconds	High	100/day
List of employees for an office	Batch	15 minutes	Medium	5/week
Show salary grade for all jobs	Online	6 seconds	Low	5/week

Identifying Access Requirements

You identify access requirements for each transaction by analyzing the business functions documented during the logical design process. Different business functions require different access to the database.

Business function

The following business function specifies that you need to access the SKILL, EXPERTISE, and EMPLOYEE entities:

Add a skill for an employee.

Sample transactions

The following table shows access requirements for three sample transactions.

Transaction	Processing Mode	Time	Priority	Frequency of Access	Access Requirements
Add or delete a claim	Online	3 seconds	High	100/day	EMPLOYEE CLAIM
List of employees for an office	Batch	15 minutes	Medium	5/week	OFFICE EMPLOYEE
Show salary grade for all jobs	Online	6 seconds	Low	5/week	JOB SALARY GRADE

Determining the Database Entry Point and Access Key for Each Transaction

You need to determine the first entity that each transaction accesses in the database. Identifying entry points can point out the need for additional indexes, or, as will be seen in Determining How an Entity Should Be Stored, the need for an entity to be stored with a location mode of CALC.

You can determine the database entry point and the data element used as an **access key** for a transaction by reviewing the access path diagram that you developed for the transaction during the logical design process. Specify the name of the entity *and* the data element used to access the entity.

Sample transactions

The following table shows the database entry points and access keys for three typical transactions.

Transaction	Processing Mode	Time	Priority	Frequency of Access	Access Requirements	Entry Point
Add or delete a claim	Online	3 seconds	High	100/day	EMPLOYEE CLAIM	EMPLOYEE (EMP ID)
List of employees for an office	Batch	15 minutes	Medium	5/week	OFFICE EMPLOYEE	OFFICE (OFFICE CODE)
Show salary grade for all jobs	Online	6 seconds	Low	5/week	JOB SALARY GRADE	JOB (None)

Projecting Growth Patterns

Projecting the minimum, most frequent, and maximum number of entity occurrences helps you to determine how much space is required to support a database. These projections should be for a specified period of time.

To structure the database correctly, you need to make the following projections for each entity:

- Minimum number of occurrences—Identifies the starting point for the database and, when compared to the maximum, gives you an idea of the projected growth.
- Most typical number of occurrences—Identifies the number of occurrences seen most frequently in the database (the mode). This number is used in determining the number of entity occurrences in a relationship and during performance analysis.
- Maximum number of occurrences—Identifies the largest expected number of occurrences of this entity. This figure is used for sizing the database.

Sample number of entity occurrences

The following table shows the projected number of occurrences for each entity in the Commonweather Corporation database.

Entity Name	Minimum	Most Frequent	Maximum
DEPARTMENT	9	15	20
EMPLOYEE	560	1000	1500
OFFICE	36	90	150
JOB	41	80	120
SKILL	68	80	120
STRUCTURE	1000	1500	2000
EMPOSITION	2000	2500	3000
EXPERTISE	3000	3500	4000
COVERAGE	1000	4000	6000
LIFE INS PLAN	3	4	5
HEALTH INS PLAN	5	10	10
INS CO	5	10	15
HOSPITAL CLAIM	800	3000	5000
NON-HOSPITAL CLAIM	1000	4000	6000
DENTAL CLAIM	2500	5000	7000
PATIENT	2000	5000	7000

DENTIST	100	300	1000
PROJECT	350	500	1000
NON-HOSPITAL PROCEDURE	2000	5000	8000
DENTAL PROCEDURE	4500	7000	9000
PHYSICIAN	100	300	1000
HOSPITAL	50	100	300
WORKER	560	3000	5600

Determining the Number of Entities in Each Relationship

To determine the sizing characteristics of the database, you will need to know the number of entities in each data relationship. For example, you will need to know the number of employees in each department to allow for effective placement of the EMPLOYEE and DEPARTMENT database entities.

Document both the expected and maximum number of entities in each relationship. If these numbers cannot be provided, use the statistics on numbers of entity occurrences gathered earlier to determine the numbers. For example, you can calculate the maximum number of employees in each department by dividing the maximum number of EMPLOYEE entity occurrences by the maximum number of DEPARTMENT entity occurrences.

Sample numbers of relationship entity occurrences

The following table shows the projected number of entity occurrences in three sample data relationships.

Relationship	Expected	Maximum
Employees in each department	66	75
Employees in each office	8	20
Positions for each employee	2	5

Determining How Often Each Entity Will Be Accessed

If you know how often each entity will be accessed, you will be able to predict potential bottlenecks in the system. To estimate how frequently each entity will be accessed:

- Review the database access path of each transaction that uses the entity.
- Analyze the frequency with which each transaction will be executed.

Sample entity access rates

The following table shows how often three sample database entities might be added, deleted, updated, or retrieved in the course of business at Commonweather Corporation.

Entity Name	Adds	Deletes	Updates	Reads
DEPARTMENT	3/year	3/year	1/week	25/day
EMPLOYEE	4/month	3/month	8/week	100/day
JOB	1/week	1/week	5/week	25/day

Chapter 11: Determining How an Entity Should Be Stored

This section contains the following topics:

Overview

You have now created a preliminary data structure diagram (Chapter 9, "Creating a Preliminary Data Structure Diagram") and have gathered the information necessary to refine this diagram (Chapter 10, "Identifying Application Performance Requirements"). This chapter discusses the first step in the refinement process: assigning location modes to the entities in the database.

Location Modes

To guarantee efficient database performance, you need to plan the best use of computer storage resources and provide for the most efficient data access. Several facilities are available under CA IDMS/DB for this purpose. By minimizing the number of input/output operations performed against the database, these facilities ensure optimal processing performance.

The data location modes in CA IDMS/DB provide you with the following capabilities:

- Randomization
- Clustering

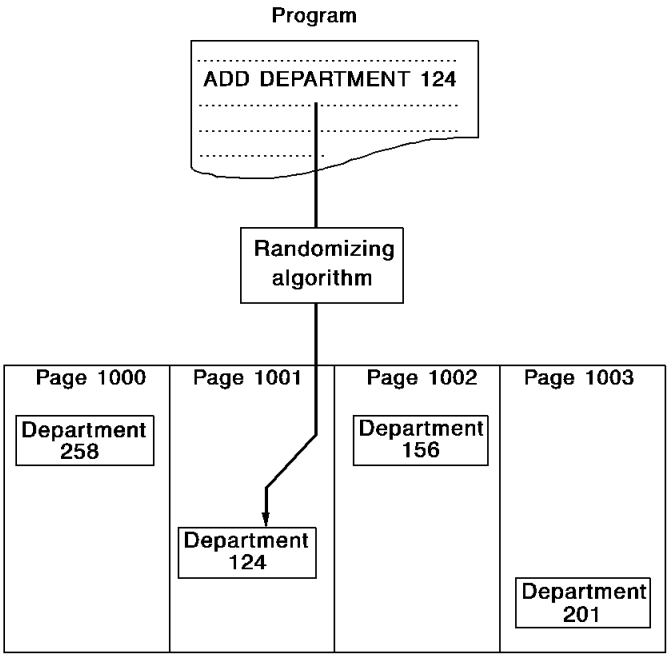
Randomization

CALC location mode

CA IDMS/DB allows users to distribute occurrences of a particular entity randomly across the area to which it is assigned. Randomization of entity occurrences is achieved with the **CALC location mode**.

When you specify CALC for an entity, the database uses a randomizing algorithm to calculate a storage page for each occurrence of that entity; the calculation is based on the value of a symbolic key (called the CALC key).

The diagram below shows the use of the CALC location mode to randomize entity occurrences.



CA IDMS/DB stores an occurrence of a CALC entity on or near a calculated storage page. The entity is placed directly on the preferred page if sufficient space exists. Otherwise, it is placed on the next page within the area where sufficient space exists. If the end of the area is reached in the search for space, CA IDMS/DB wraps around to the beginning of the area.

Purpose of the CALC location mode

The purpose of the CALC location mode is twofold:

- **Direct retrieval by symbolic key**, enabling retrieval of an entity occurrence with a single read operation. Retrieval of an entity located CALC involves knowing only the value of its CALC key; the database automatically converts the CALC key into the correct page number when the entity is requested. For more information concerning the use of numeric fields within a record's CALC key, see Zoned and Packed Decimal Fields as IDMS Keys.
- **Random distribution of entity occurrences** over all the pages in an area. This reduces overflow conditions and leaves space for clustered entity occurrences. For further information on overflow conditions, see "Overflow Conditions" in Chapter 15, "Determining the Size of the Database".

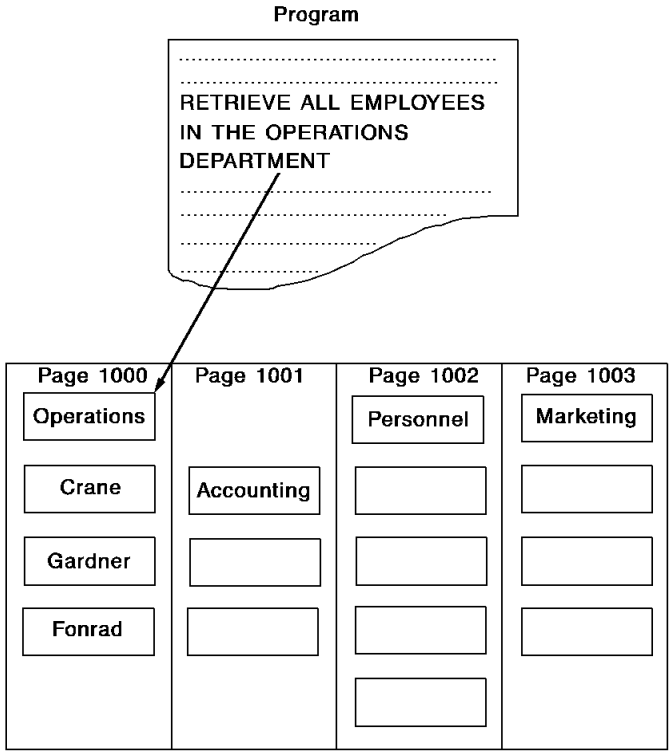
Clustering

Clustering enables you to group entity occurrences that are likely to be accessed together. When you request clustering, the database stores each entity occurrence as close as possible to another occurrence to which it is logically related.

Minimizing read operations

By storing related entity occurrences on or near the same page, clustering minimizes the number of read operations required to access the database. Clustering could, for example, be used to retrieve a DEPARTMENT entity occurrence and its related EMPLOYEE entity occurrences with a single read operation.

Clustering enhances processing performance by grouping entity occurrences that are likely to be accessed together. For example, clustering could be used to store employees CRANE, GARDNER, and FONRAD on the same database page as the OPERATIONS department, the department to which these employees belong. All four entity occurrences could be retrieved with a single read operation.



Clustering methods

CA IDMS/DB supports the following methods of clustering entity occurrences:

- Clustering through a relationship allows you to cluster entity occurrences related through a relationship. This causes an entity (the **child**) to be stored as close as possible to the entity it references (the **parent**).

  If assigned to the same area, child occurrences will target to the same page as their parent.

  When assigned to a different area, child occurrences are stored at the same relative position in their area as the parent occurrence is in its area.

  This is the most efficient means of clustering two or more related entities.

  To indicate clustering through a relationship, you specify a location mode of CLUSTERED and the name of the relationship around which this entity is to be clustered.

  For further information on how CA IDMS/DB clusters entity occurrences, see the *CA IDMS Database Administration Guide*.

- Clustering through an index allows you to cluster entity occurrences based on the value of a symbolic key. If clustering using an index, all occurrences having the same (or similar) index key values are targeted to the same database page. This has the effect of maintaining entity occurrences physically in sequence by the value of the key.

  This is the most efficient means of ordering data occurrences if multiple occurrences are often retrieved in the sequence of their key values. However, its benefit is minimized if frequent additions and deletions cause entity occurrences to be stored out of sequence due to overflow conditions.

  To indicate clustering through an index, you specify a location mode of CLUSTERED and the name of the index around which this entity is to be clustered.

  For more information on indexes, see Refining the Database Design.

- Clustering using the CALC location mode allows you to cluster entities related through a shared data element. You assign the CALC location mode to each entity, defining corresponding data elements as CALC keys.

  When the CALC location mode is specified for two entities, CA IDMS/DB stores all entity occurrences that have the same CALC key value on or near the same database page.

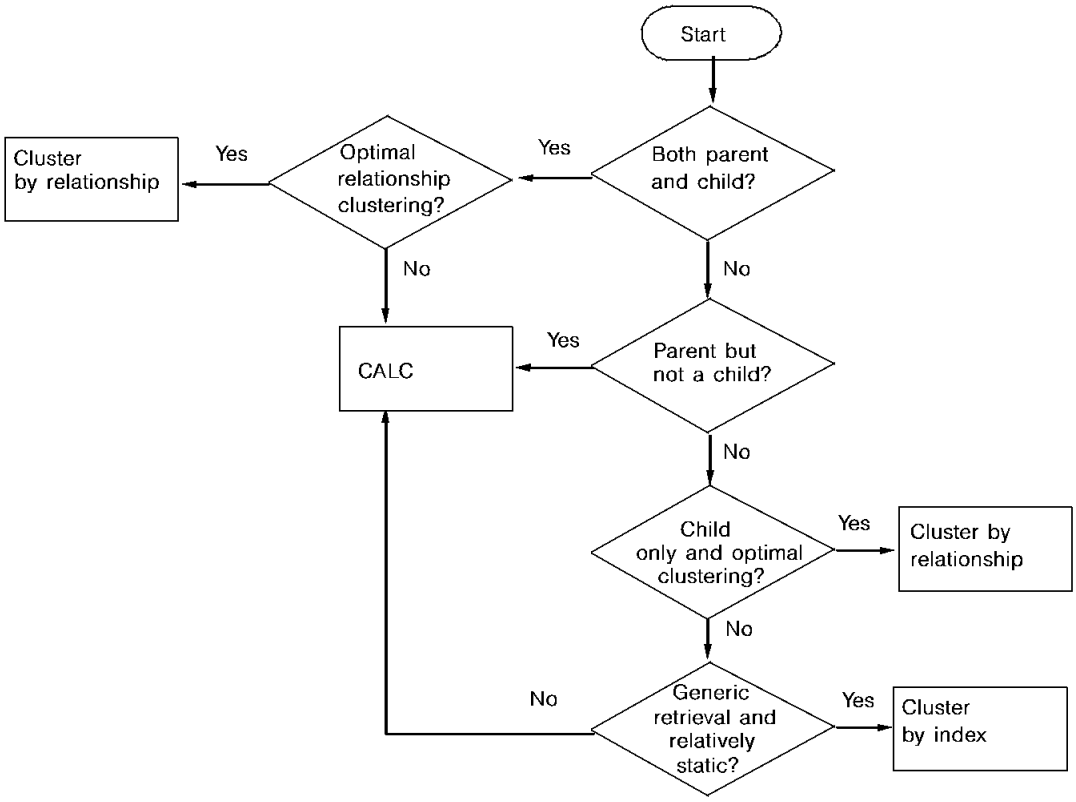
  This is a means of clustering entities even if no relationship exists but does not work well for extremely volatile or high-volume entities. Frequent additions and deletions of entity occurrences may increase the likelihood of contention and, if many occurrences target to the same page, overflow conditions will increase I/O rates.

  To indicate clustering using the CALC location mode, you specify a location mode of CALC for each entity, defining identical data elements as CALC keys.

A discussion of when to choose these methods follows.

Guidelines for Determining How an Entity Should Be Stored

Guidelines for assigning location modes to entities are shown below. As you determine how you want to store each entity, indicate this information on your data structure diagram.



The decision operations in the chart are discussed below, followed by a discussion of how to assign data location modes to entities in the Commonweather Corporation database.

Is This Entity Both a Parent and a Child?

Ask this question of every entity identified in the logical database design.

If the answer to this question is **Yes** for an entity, the entity is involved in multiple relationships and you must decide which, if any, of these relationships should be used for clustering.

Is There Optimal Relationship Clustering for This Entity?

If the entity is involved in multiple relationships in which it is both the parent and child, it may be possible to cluster this entity around another related entity. **Optimal clustering** means that application programs access this entity most often in conjunction with another entity and clustering can be used effectively.

Clustering through a relationship is one of the most effective ways of reducing I/Os when related entity occurrences are retrieved together. Therefore, if applications accessing this entity frequently access related entities, you should generally cluster the child entities through the relationship.

Note: If the size of all clustered entity occurrences is too large, the benefit of clustering might be negated because several I/Os are required to access the entire cluster.

If there is no optimal clustering, the entity should be stored CALC, providing both an alternate entry point into the database and a parent around which other entities can be clustered.

Example

An example of such an entity is the EMPLOYEE entity. This entity is both a parent and a child but has no optimal clustering.

The COVERAGE entity, on the other hand, is both a parent and child but can be clustered optimally around the EMPLOYEE-COVERAGE relationship since access is most often by means of the EMPLOYEE entity, and multiple COVERAGE entity occurrences relating to a particular employee are often accessed at the same time.

Is This a Parent Entity but Not a Child Entity?

Ask this question for each entity that does not exist as both a parent and a child.

An entity that exists only as a parent entity is often used as an entry point into the database. For this reason, it is advisable to have a fast access key on the entity.

The CALC location mode generally is a better choice than an index key because:

- It requires fewer I/Os to access an entity using a CALC key.
- The CALC algorithm randomizes entity occurrences, thus allowing space to cluster related entity occurrences.

Example

An example of a parent entity but not a child entity is the DEPARTMENT entity. This entity should be stored CALC based on the DEPT ID.

Is This a Child Entity but Not a Parent Entity?

Ask this question of each entity that exists neither as a parent and child, nor as only a parent.

An entity that acts as a child but not a parent is not usually used as an entry point into the database. This entity often can be stored clustered around one of its parent entities.

Clustering through a relationship is one of the most effective ways of reducing I/Os when related entity occurrences are retrieved together. Therefore, if applications accessing this entity frequently access related entities, you should generally cluster the child entities through the relationship.

Note: If the size of all clustered entity occurrences is large, the benefit of clustering might be negated because it requires several I/Os to access the entire cluster.

Example

An example of a child entity but not a parent is the EXPERTISE entity. An occurrence of this entity is most frequently accessed through its associated EMPLOYEE entity occurrence. Therefore, it can be stored clustered around the EMP-EXPERTISE relationship.

Is Generic Retrieval Required and Is the Entity Relatively Static?

The only entities left to ask this question of are standalone entities and child-only entities having no optimal clustering.

You should choose CALC location mode if application programs always retrieve this entity using its full key or if it is relatively dynamic (that is, many additions, deletions, or key changes).

If an entity is relatively static and multiple occurrences are often retrieved together, it is most effective to cluster the entity through an index defined on the most-commonly used access key.

If the entity is not static, but often participates in multi-occurrence retrievals, cluster the entity on an index defined on its db-key. For more information on indexes, Chapter 12, "Refining the Database Design"

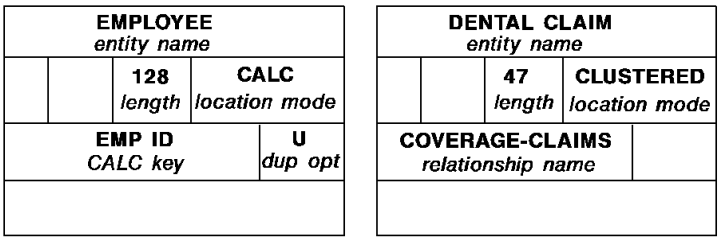
Graphic Conventions

There are graphic conventions used to represent both the location mode and indexes.

Conventions for Specifying Location Mode

To indicate your location mode decision on the data structure diagram, you need to name the method (CALC or CLUSTERED). If the entity is to be stored CALC, name the CALC key. If the entity is to be clustered, name the relationship or the index it is to be clustered around.

The diagram below shows how your location method decisions are indicated on the diagram. The EMPLOYEE entity has a location mode of CALC. Its CALC key is the data element EMP ID and duplicates of this key are not allowed; the key must be unique. The second example is the DENTAL CLAIM entity, which has a location mode of CLUSTERED. Occurrences of this entity will be clustered around the COVERAGE-CLAIMS relationship.



The following characteristics of the entities are indicated on the diagram:

- *Entity name*— The name of the entity
- *Length*— The estimated data length (in bytes) for fixed-length entities; the maximum length for variable-length entities. This information is used in database sizing.
- *Location mode*— How the entity is stored in the database (CALC or CLUSTERED).
- *CALC-key, relationship name, or index name*—The name of the CALC-key field (CALC entities) or the name of the relationship around which this entity is to be clustered (if the entity is to be clustered around a relationship), or the name of the index around which this entity is to be clustered (if the entity is to be clustered around an index).
- *Dup opt* (CALC entities only)—The duplicates option: the disposition of entities with duplicate CALC keys (**U** for unique or blank for non-unique).

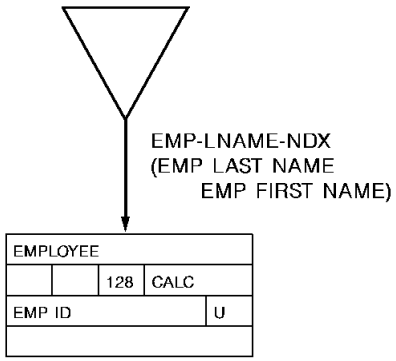
Conventions for Representing Indexes

To represent an index on the data structure diagram:

- Use a triangle to represent the index.
- Specify a name for the index.
- Identify the data element name(s) that are to be indexed.
- Specify whether duplicate indexed keys are allowed (blank) or not allowed (**U**).

Sample index representation

The following diagram shows the standard CA IDMS/DB notation for an index. The index allows the DBMS to access all EMPLOYEE entity occurrences in the database based on the last name/first name in ascending order. Duplicate last name/first name combinations are allowed.



Location Modes for Entities in the Commonweather Database

By following the guidelines presented in this chapter, you can assign appropriate location modes to the entities in your database. The table below shows how the location mode was decided upon for each entity in the Commonweather database.

Is this entity...	Both parent and child?	With optimal clustering?	Parent and not child	Child and not parent (w/optimal clustering)?	Generic retrieval and relatively static?
DEPARTMENT	N	-	Y	-	-
OFFICE	N	-	Y	-	-
PROJECT	Y	N	-	-	-
INS CO	N	-	Y	-	-
LIFE INS PLAN	N	-	N	N	Y

HEALTH INS	Y	N	-	-	-
PLAN	Y	N	-	-	-
NON-HOSPITAL	N	-	N	N	-
CLAIM	Y	N	-	-	-
DENTAL CLAIM	N	-	Y	-	-
HOSPITAL	Y	N	-	-	-
PHYSICIAN	N	-	Y	-	-
DENTIST	N	-	Y	-	-
EMPLOYEE	N	-	Y	-	-
JOB					
SKILL					
PATIENT					

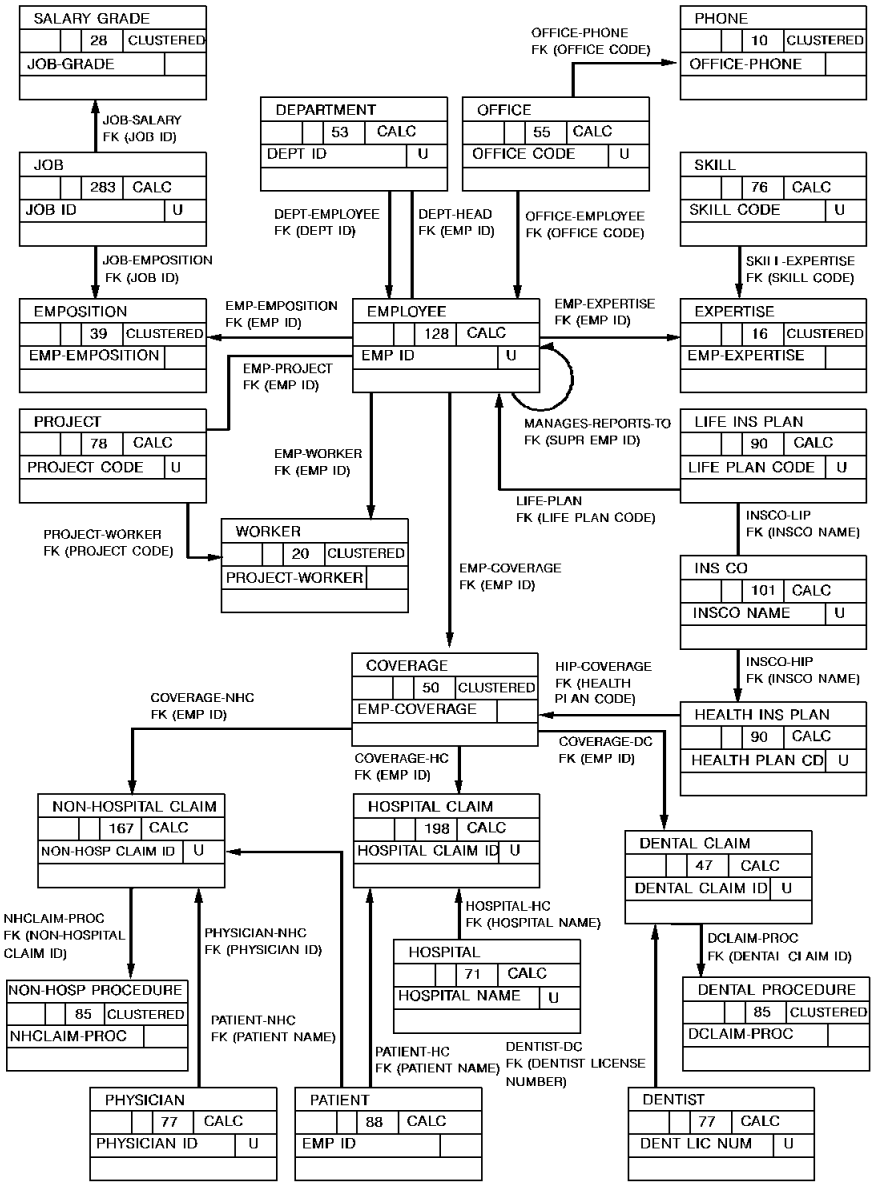
Location mode: Store CALC on primary key. For example, store the EMPLOYEE entity CALC on EMP ID.

Is this entity...	Both parent and child?	With optimal clustering?	Parent and not child	Child and not parent (w/optimal clustering)?	Generic retrieval and relatively static?
EMPOSITION	N	-	Y	-	-
EXPERTISE	N	-	Y	-	-
STRUCTURE	Y	N	-	-	-
WORKER	N	-	Y	-	-
PHONE	N	-	N	N	Y
SALARY	Y	N	-	-	-
GRADE	Y	N	-	-	-
COVERAGE	N	-	N	N	-
NON-HOSPITAL	Y	N	-	-	-
PROCEDURE	N	-	Y	-	-
DENTAL	Y	N	-	-	-
PROCEDURE	N	-	Y	-	-

Location mode: Store clustered on the optimal relationship. For example, store the EXPERTISE entity clustered on the EMP-EXPERTISE relationship

Revised Data Structure Diagram for the Commonweather Corporation

After you have decided how you want to store and access each entity, indicate this information on the data structure diagram. Below is the updated data structure diagram for the Commonweather Corporation database.



Chapter 12: Refining the Database Design

This section contains the following topics:

Evaluating the Database Design

You have created a preliminary model for a physical database and have identified the entities in the database. You have also gathered the information necessary to refine this diagram and have assigned location modes to the entities. Now you will refine the preliminary design to allow for optimal transaction and system performance.

Evaluation considerations

Before you refine the data structure diagram, you need to evaluate the design for performance. To satisfy performance requirements for each individual business transaction, you need to consider the following issues:

- **Input/output (I/O) performance**—Is the number of I/O operations performed against the database sufficiently low to provide satisfactory transaction performance?
- **CPU time**—Does the structure of the physical database optimize the use of CPU processing?
- **Space management**—Do design choices help to conserve storage resources?

Once you have refined the database to satisfy each individual transaction, you need to determine how the system will be affected by the concurrent execution of several transactions. To avoid excessive **contention** for database resources, you need to make appropriate changes to the physical model.

Refining the database design

Like many other database design procedures, refining the database design is an iterative process, as shown below. As you refine the design, you need to evaluate the design for performance. When you make changes, you should review the design to ensure that it will optimize processing for all critical transactions and also minimize the likelihood of contention.

Refinement Options

CA IDMS/DB provides options for refining the database design to ensure optimal performance in individual transactions. *There is no right or wrong method for refining the physical database model*. Your organization's requirements will determine the best approach for you.

Options

The following database options can be used to ensure optimal performance in individual business transactions:

- **Indexes**—Chapter 11, "Determining How an Entity Should Be Stored" showed you how to include indexes in the database design to provide data clustering. At this point in the design process, you have the option to include additional indexes to provide generic search capabilities as well as alternate access keys.
- **Collapsing relationships**—A one-to-many relationship can be expressed within a single entity by making the many portion of the relationship a repeating data element. A one-to-many relationship expressed in this way can enhance processing performance by reducing DBMS overhead associated with processing multiple entity occurrences.
- **Introducing redundancy**—By maintaining certain data redundantly, you can sometimes enhance processing efficiency in selected applications.

Each of these options is described in detail below following a discussion of how to estimate I/Os for transactions.

Estimating I/Os for Transactions

After you have assigned data location and access modes to the entities in a database, you need to estimate the number of input/output operations that each business transaction will perform. You estimate the I/O count for a transaction by tracing the flow of processing from one entity to another in the database. As you trace the flow of processing, you determine the number of I/Os required to access all necessary entities.

The I/O estimate for a business transaction depends on several factors, including:

- The order in which entities are accessed
- The location mode of each entity accessed
- The types of indexes (if any) used to access the data
- How the entities are clustered in the database

General guidelines

Assuming that an entire cluster of database entities can fit on a single database page, you can use the following general guidelines for estimating I/Os:

- Zero I/Os are required to access an entity that is clustered around a previously accessed entity.
- One I/O is required to access an entity stored CALC.
- Three I/Os are required to access an entity through an index.

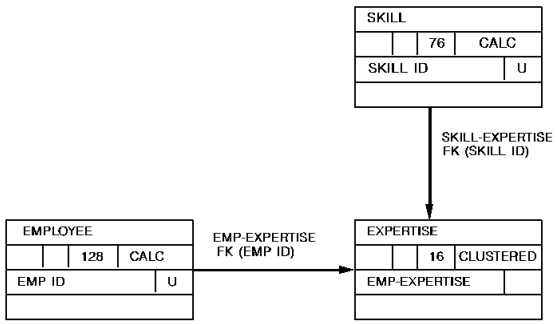
To calculate the time required to perform all I/O operations in a particular transaction, perform the following computations:

- **Total number of I/Os for all entity types**—Compute the total number of I/O operations by adding the number of I/Os required to retrieve and update occurrences of all entity types.
- **I/O reserve factor**—Multiply the total number of I/Os by 1.5 to account for possible overflow conditions and large index structures.
- **Amount of time to perform I/Os**—Multiply the total number of I/Os for all entity types by the access time for the device being used. The result is a rough estimate of the time required to perform all I/O operations in the transaction.

Once you have determined how much time will be required to execute a particular transaction, you need to compare this time figure with the performance goal you established earlier in the design process. If the required time does not meet your expectations, you need to modify the physical database model until it does. Sometimes you have to change your expectations.

For further information on establishing performance goals for business transactions, see Chapter 10, "Identifying Application Performance Requirements".

Two sample exercises in estimating I/Os are presented below. Each exercise uses the EMPLOYEE, EXPERTISE, and SKILL entities:



Sample Exercise #1: Estimating I/Os for a Retrieval Transaction

Suppose you need to estimate I/Os for the following transaction:

Identify skills for an employee.

In this transaction, the user specifies an employee ID value and the system returns the employee ID, name, skill code, skill level, and skill description for the specified employee. This transaction uses the EMPLOYEE entity as an entry point to the database.

I/O estimates

By analyzing the access path of the transaction, you can make the following I/O estimates for each entity accessed:

- **EMPLOYEE**—Because this entity is stored CALC, only one I/O operation is required to retrieve one EMPLOYEE entity occurrence from the database.
- **EXPERTISE**—Each employee can have as many as five skills. Therefore, the transaction retrieves five EXPERTISE entity occurrences for each EMPLOYEE entity occurrence. However, since EXPERTISE entity occurrences are clustered around a related EMPLOYEE entity occurrence, no I/Os are necessary to retrieve the EXPERTISE entity occurrences.
- **SKILL**—For each EXPERTISE entity occurrence retrieved, there is an associated SKILL entity occurrence in the database. Therefore five SKILL entity occurrences are retrieved for each employee. Since the SKILL entity is stored CALC, its occurrences are distributed randomly in the database. To retrieve five SKILL entity occurrences, the system must perform five I/Os.

Estimating I/Os for a sample retrieval transaction

A total of six I/O operations will be performed by this transaction, as shown below.

	Identify skills for an employee	Number of I/Os to access one occurrence	Number of occurrences accessed			Total I/Os for entity type
			Read	Write	Total	
Record	*Employee*	*1*	*1*		*1*	*1*
Record	*Expertise*	*0*	*5*		*5*	*0*
Record	*Skill*	*1*	*5*		*5*	*5*
Record						
Record						
Record						
Record						

Total number of I.Os for the transaction	*6*
Total I/Os plus reserve factor of 50%	*9*
Minimum time for the transaction (I/Os * .025 sec)	*.225*

Sample Exercise #2: Estimating I/Os for an Update Transaction

When you estimate I/Os for a transaction that performs update functions, you need to consider I/O operations that must be executed to ensure database integrity. In addition to the I/Os required to access desired entities, update transactions must perform I/Os to access related entities. Some types of integrity checking require that the system access other related entities.

Suppose you need to estimate I/Os for the following transaction:

Add a skill for an employee.

To protect the relationship between an EMPLOYEE entity and an associated EXPERTISE entity, the EMPLOYEE entity must be accessed before storing the EXPERTISE entity. Likewise, to protect the relationship between a SKILL entity and an associated EXPERTISE entity, the SKILL entity must be accessed before storing the EXPERTISE entity.

I/O estimates

Knowing this information, you can make the following I/O estimates for each entity accessed:

- **EMPLOYEE**—Because this entity is stored CALC, only one I/O operation is required to access one EMPLOYEE entity in the database.
- **SKILL**—Since the SKILL entity is stored CALC, only one I/O is required to access a single SKILL occurrence in the database.
- **EXPERTISE**—EXPERTISE entities are clustered around a related EMPLOYEE entity. Therefore one I/O is necessary to store the EXPERTISE entity.

Estimating I/Os for a sample update transaction

A total of three I/O operations will be performed by this transaction, as shown below.

	Add a skill for an employee	Number of I/Os to access one occurrence	Number of occurrences accessed			Total I/Os for entity type
			Read	Write	Total	
Record	*Employee*	*1*	*1*		*1*	*1*
Record	*Skill*	*1*	*1*		*1*	*1*
Record	*Expertise*	*0*	*0*	*1*	*1*	*1*
Record						
Record						
Record						
Record						

Total number of I.Os for the transaction	*3*
Total I/Os plus reserve factor of 50%	*4.5*
Minimum time for the transaction (I/Os * .025 sec)	*.113*

Eliminating Unnecessary Entities

Sometimes entities identified during the logical design are not required as separate entities in the physical implementation. Two ways to eliminate such entities are:

- Collapsing relationships
- Introducing redundancy

Collapsing Relationships

During the normalization process in logical database design, you separated multiply-occurring data into a separate entity type (first normal form). It may be more efficient to move this data back into the original (parent) entity.

Consider this option if data occurs a fixed number of times and the data is not related to another entity. An example of such data is monthly sales totals for the last twelve months collapsed into a sales entity.

Advantages

By maintaining the data in a single entity instead of maintaining two separate entity types, you can:

- Save storage space that might otherwise be used for pointers or foreign-key data.
- Reduce database overhead by eliminating the need to retrieve two entities. When you express a one-to-many relationship within a single entity, application programs can access all desired data with a single DBMS access.

Note: Expressing a one-to-many relationship within a single entity offers little I/O performance advantage over clustering two separate entities.

Comparison of collapsing relationships and maintaining separate entities

The following table presents a comparison of collapsing relationships into a single entity type and maintaining separate entities.

Efficiency Considerations	Potential Impact
I/O	Expressing a one-to-many relationship within a single entity offers little I/O performance advantage over clustering two entities.
CPU time	By storing a repeating element in an entity, you can reduce the amount of CPU time required to access the necessary data.
Space management	By storing a repeating element in an entity instead of maintaining two separate entity types, you can save storage space that might otherwise be required for pointers or foreign key data.
Contention	No difference

SQL considerations

Because repeating elements violate first normal form, they are incompatible with the relational model and cannot be defined in SQL. However, if there are a fixed number of repetitions (such as months in a year), the repeating elements can be separately named (such as JANUARY, FEBRUARY, and so on). If there is a variable but quite small number of occurrences (such as phone numbers), a fixed maximum number of elements can be named (PHONE1, PHONE2, for example), using the nullable attribute to allow identification of occurrences that might not have a value.

Introducing Redundancy

Although data redundancy should normally be avoided, you can sometimes enhance processing efficiency in selected applications by storing redundant information. A certain amount of planned data redundancy can be used to simplify processing logic.

In some instances, you can eliminate an entity type from the database design by maintaining some redundant information. For example, you might be able to eliminate an entity type by maintaining the information associated with this entity in another entity type in the database. When you *merge* two or more entity types in this way, you simplify the physical data structures and reduce relationship overhead.

Considerations

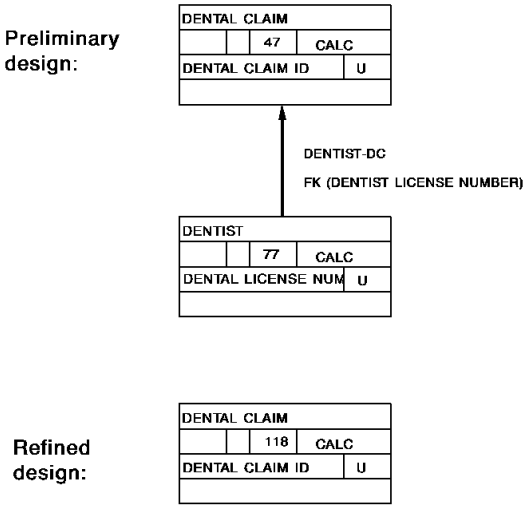
Consider maintaining redundant data under the following circumstances:

- **An entity type is never processed independently of other entity types**. If an entity is always processed with one or more additional entity types, you may be able to eliminate the entity and store the information elsewhere in the database. Since the information associated with the entity is not meaningful by itself, inconsistent copies of the data should not present a problem for the business.
- **An entity type is not used as an entry point to the database**. If application programs do not use a particular entity type as an entry point to the database, you may be able to eliminate the entity type from the design. However, do not eliminate the entity if it is a junction entity type in a many-to-many relationship.
- **The volume of data to be stored redundantly is minimal**. Do not maintain large amounts of data redundantly. A high volume of redundant information will require excessive storage space.

Example

The following diagram shows how you might use data redundancy to enhance processing of dental claim information.

By maintaining all DENTIST information with the DENTAL CLAIM entity, you can simplify the database design and reduce the overhead of maintaining the relationship. Since Commonweather users do not process information associated with the DENTIST entity by itself, inconsistent DENTIST information will not present a problem for the business.



Eliminating Unnecessary Relationships

The purpose of a relationship is to represent integrity rules between entities. As such, they serve a useful purpose in modeling your business. However, there is always overhead associated with a relationship. Since the DBMS must ensure the integrity of a relationship during update operations, they result in increased CPU and I/O. They may also require additional storage space.

While you should not sacrifice needed integrity, you should eliminate relationships that are not required for business reasons. Particularly review the need for:

- One-to-one relationships

  For example, the DEPARTMENT-HEAD relationship may not require DBMS enforcement of integrity and, if so, should be eliminated as a relationship.

- Relationships in which there are only a few pre-established parent occurrences

  Examples of this type of relationship would be STATE-OFFICE or SEX-EMPLOYEE. Ensuring that each office is in a valid state or that each employee is assigned a valid sex should be done in one of the following ways rather than as a relationship.

  - By using a map edit or code table (application enforcement)
  - By using a check constraint (in SQL-defined databases)
  - By using database procedures (in non-SQL defined databases)
  - Through a logical record facility path (in non-SQL defined databases)

In the Commonweather database, the relationship between INSCO and HEALTH INSURANCE PLAN can be removed.

Adding Indexes

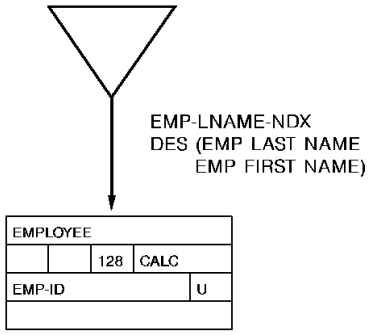
In Determining How an Entity Should Be Stored, you included indexes in the physical database model for entities that will be accessed through multi-occurrence retrievals. These entity occurrences will be clustered around the index. You now have the option to define additional indexes for database entities to satisfy processing requirements.

Review the function lists and access paths that you documented during the logical design process to ensure that each entry point entity has an efficient access for each application search key. If necessary, add additional indexes as alternate access keys to satisfy application requirements.

For further information on how to determine the database entry point for each business transaction, see Chapter 10, "Identifying Application Performance Requirements".

What is an index?

An index is a data structure consisting of addresses (db-keys) and values from one or more data elements of a given entity. Indexes enhance processing performance by providing alternate access keys to entities.



Advantages and disadvantages

While indexes minimize the number of I/Os required to retrieve data from the database, they require extra storage space and add overhead for maintenance. The addition of an index actually *increases* the I/Os and processing time required to add or remove an entity occurrence. You will need to weigh the options when considering the use of indexes.

Why add additional indexes?

Indexes provide a quick and efficient method for performing several types of processing.

- **Direct retrieval by key**—With an index, the DBMS can retrieve individual entity occurrences directly by means of a key. For example, an application programmer could use an index to quickly access an employee by social security number.

Because more than one index can be defined on an entity (each on a different data element), they can be used to implement multiple access keys to an entity.

- **Generic access by key**—Indexes allow the DBMS to retrieve a group of entity occurrences by specifying a complete or partial (generic) key value. For example, an index could be used to quickly access all employees whose last names begin with the letter M. A string of characters, up to the length of the symbolic key, can be used as a generic key.
- **Ordered retrieval of occurrences**—The DBMS can use a sorted index to retrieve entity occurrences in sorted order. In this case, the keys in the index are automatically maintained in sorted order; the entity occurrences can then be retrieved in ascending or descending sequence by key value. The application program does not have to sort the entity occurrences after retrieval. For example, all employees could be listed by name. Because entity occurrences can be accessed through more than one index, they can be retrieved in more than one sort sequence.
- **Retrieval of a small number of entity occurrences**—An index improves retrieval of all occurrences of a sparsely-populated entity and provides a way of locating all occurrences of such entities without reading every page in the area (an area sweep). Area sweeps are the most efficient means of retrieving entities with occurrences on all (or almost all) pages in an area.
- **Physical sequential processing by key**—Entity occurrences can be stored clustered around an index. With this storage mode, the physical location of the clustered entity occurrences reflects the ascending or descending order of their db-keys or symbolic keys. If occurrences of an entity are to be retrieved in sequential order, storing entity occurrences clustered via the index reduces I/O. This option is most effective when used with a stable database.
- **Enforcement of unique constraints**—An index can be used to ensure that entity occurrences have unique values for data elements; for example, to ensure that employees are not assigned duplicate social security numbers.

  Other means of enforcing unique constraints include:

  - Using a unique CALC key
  - Using a sorted relationship

Index keys

The keys associated with an index can be either:

- **Symbolic keys**, in which the key values in the index are the same as one or more data elements in the indexed entity occurrences
- **Db-keys**, in which the key values in the index are the db-keys of the indexed entity occurrences.

Symbolic key indexes are useful for:

- Enforcing unique constraints

- Providing alternate access keys (entry points) into the database
- Supporting generic and ordered retrieval of entity occurrences

Db-keys are useful for:

- Retrieving all occurrences of a sparsely-populated entity (an entity with occurrences on only some of the pages in an area)

If generic or ordered retrieval is not a consideration when adding new symbolic key index and the key is made up of more than one data element, choose as the first data element one which is not already an access key into the database. For example, if you place an index on COVERAGE to ensure that its primary key is unique, then the index key will be composed of: EMP ID, HEALTH PLAN CODE, and COVERAGE TYPE. Since EMP ID and HEALTH PLAN CODE are already entry points into the COVERAGE entity (because they are CALC keys of related entities), choose COVERAGE TYPE as the first data element in the index key.

Index order

The index order is the way in which the entity occurrences will be logically ordered based on the key or keys you have chosen. Index orders include:

- **Ascending**—Index entries are ordered so that an entry with a lower key value occurs before an entity with a higher key value: A through Z, smallest to largest.
- **Descending**—Index entries are ordered so that an entry with a higher key value occurs before an entity with a lower key value: Z through A, largest to smallest.
- **Mixed**—You can choose to have one key of an index ordered in one order and another key of the same index in a different order.

In general, choose an index order based on how data is most frequently accessed. For example, if employees are most often retrieved in ascending order by last name, then choose ascending as the index order.

Db-key indexes

You can choose to have the index order based on the db-keys of the entity occurrences being indexed.

Indexes ordered by db-key especially improve retrieval of entities with occurrences on only some of the pages in an area, but which are likely to have more than one occurrence per page, such as entities clustered around a sparsely occurring parent.

Retrieving all occurrences of an entity

The following table provides guidelines for choosing a retrieval method (and, thus, a design) to retrieve all occurrences of an entity.

Data in the Database	Access Method
Sparsely populated	An index based on symbolic key
Every page contains one or more occurrences of the entity	Use an area sweep
Sparsely populated but a page contains multiple occurrences of the entity	An index based on db-key

SQL considerations

In the SQL environment, every entity that is a parent in a relationship must have a unique index or CALC key defined for the referenced (primary) key. Add any indexes that are missing.

Every entity defined in an SQL-defined database is initially assigned a **default index**. This is an index sorted by db-key so that all entity occurrences can be accessed with the minimum number of I/Os. You must decide whether to retain this index or drop it. You should **drop** the default index if any of the following are true:

- The entity is densely populated; every page contains at least one occurrence of the entity.
- Entity occurrences are clustered around another index.
- Another index is defined on the entity, and it is unlikely that more than one entity occurrence resides on a page.
- Non-keyed queries will be extremely rare.

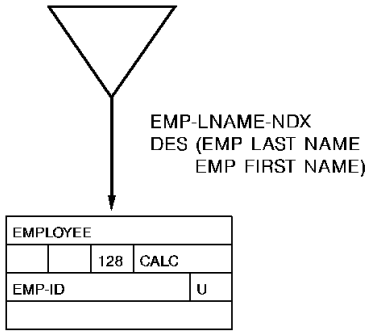
Representing additional index options

In Determining How an Entity Should Be Stored, you saw how to represent an index.

To represent additional index options in the data structure diagram:

- Specify the order for each data element used as an index key (ASC - ascending; DES - descending).
- If the order is by db-key, specify DBKEY.

The following diagram shows the standard CA IDMS/DB notation for an index. The index allows the DBMS to access all EMPLOYEE entity occurrences in the database based on the last name/first name in descending order. Duplicate last name/first name combinations are allowed.



Summary of indexes

Indexes should be added, if necessary, when validating transaction performance. Add additional indexes if the advantage gained outweighs the cost.

The following table presents a comparison of the use of indexes and user-written sort routines.

Efficiency Considerations	Potential Impact
I/O	I/O may be reduced for retrieval but increased for update.
CPU time	CPU can be reduced for retrieval but increased for update.
Space	Indexes require extra storage space in the database.
Contention	The use of an index can sometimes cause contention.

Refined Data Structure Diagram for Commonweather Corporation

Collapse relationships

You can eliminate unnecessary entities by embedding their data in a related entity type. By using a repeating data element instead of maintaining two separate entities, you can save storage space and also reduce CPU needed to access the repeating data as described below:

- The PHONE and SALARY GRADE records are ideal candidates for elimination because:
  - Each entity participates in only one relationship. The PHONE entity is related only to the OFFICE entity; the SALARY GRADE entity is related only to the JOB entity.
  - A maximum number of repetitions is predictable for each entity. A maximum of three phone numbers exists for each office; a maximum of four salary grades exists for each job.

  Thus we can eliminate the PHONE entity and place three PHONE NUMBER data elements in the OFFICE entity. We can also eliminate the SALARY GRADE entity and place four SALARY GRADE data elements in the JOB entity. If you define this database using SQL statements, each of the repeating data elements must have a unique name and, in the case of PHONE NUMBER and SALARY GRADE, allow null values.

Introduce redundancy

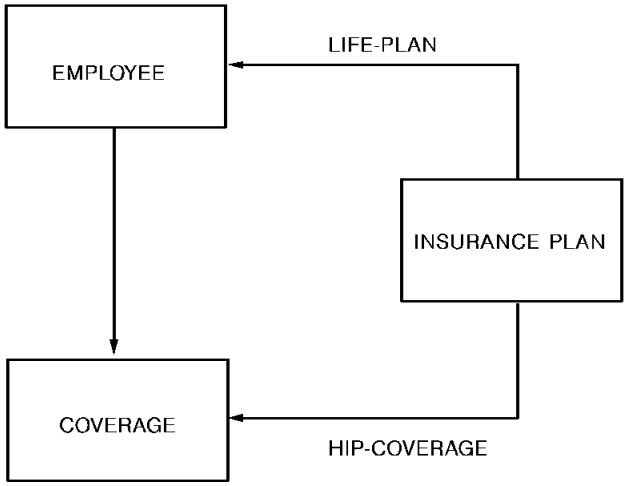
The PHYSICIAN, HOSPITAL, PATIENT, DENTIST, and INS CO entities are never processed independently of other entity types. Therefore, they do not need to be maintained independently in the database. In addition, information in the PROJECT and WORKER entities is already carried in the STRUCTURE entity. HEALTH INS PLAN and LIFE INS PLAN contain the same type of information and can be combined into a single entity. Information maintained in these entities can therefore be embedded in other related entities:

- INS CO information can be stored in HEALTH INS PLAN and LIFE INS PLAN.
- PHYSICIAN information can be maintained in NON-HOSPITAL CLAIM.
- HOSPITAL information can be maintained in HOSPITAL CLAIM.
- PATIENT information can be maintained in NON-HOSPITAL CLAIM, HOSPITAL CLAIM, and DENTAL CLAIM.
- DENTIST information can be maintained in DENTAL CLAIM.
- HEALTH INS PLAN and LIFE INS PLAN can be combined into one entity called INSURANCE PLAN.

Update anomalies for these entities will not present a problem for the organization. For example, since Commonweather users do not process DENTIST information by itself, inconsistent information in this entity will not compromise integrity or complicate business processing functions.

Eliminate unnecessary relationships

At this point, the health-related entities can be represented as:



The LIFE-PLAN relationship can be eliminated by treating it as another type of coverage available through an insurance plan. Although this change will require that an occurrence of COVERAGE be associated with each EMPLOYEE, it simplifies the database structure and the application processing.

The HIP-COVERAGE relationship can be eliminated also. Since there will never be more than 15 insurance plans in the database, the validity of an employee's insurance information (the plan code) can be enforced through other means such as a logical record facility path or an SQL CHECK constraint.

Also eliminate the DEPT-HEAD relationship. Integrity enforcement by the DBMS for this one-to-one relationship is not critical to Commonweather Corporation.

Add indexes

Add the following indexes to enforce unique constraints:

- An index on SKILL based on SKILL NAME
- An index on COVERAGE based on COVERAGE TYPE, PLAN CODE, and EMP ID
- An index on EMPOSITION based on JOB ID and EMP ID
- An index on EXPERTISE based on SKILL CODE and EMP ID
- An index on NON-HOSP PROCEDURE based on NON-HOSP CLAIM ID and PROCEDURE NUMBER

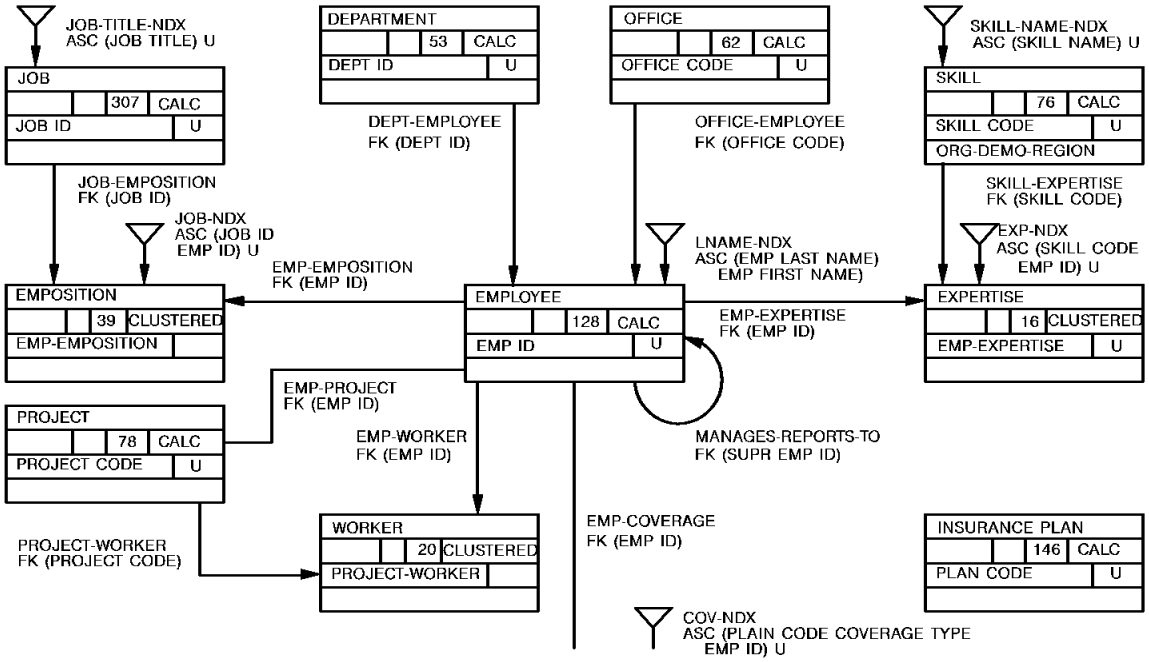
- An index on DENTAL PROCEDURE based on DENTAL CLAIM ID and PROCEDURE NUMBER

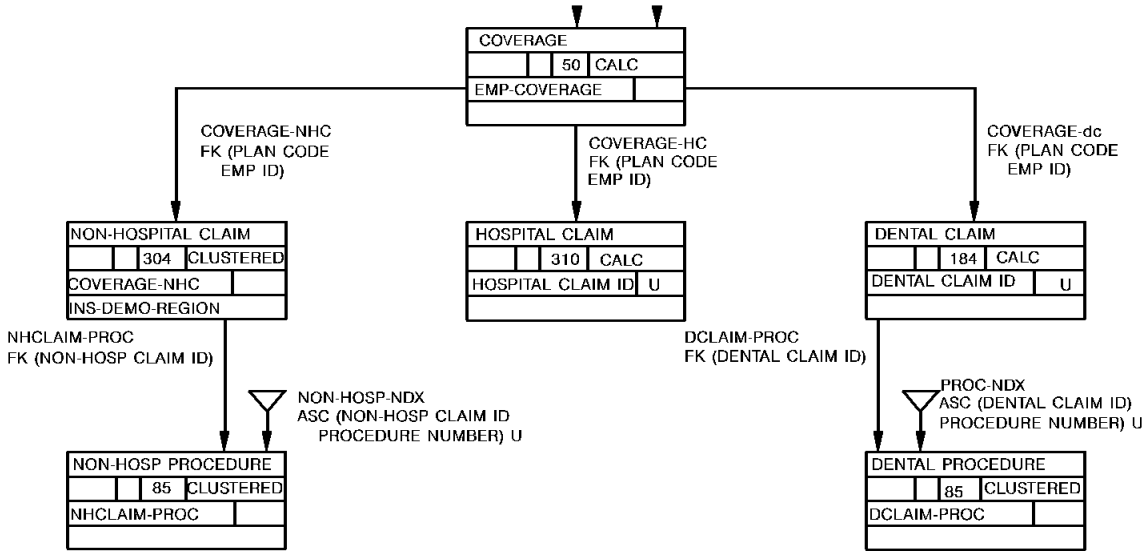
Note: You will see in the next chapter how some of these indexes can be eliminated.

Add the following indexes to provide generic search capability:

- An index on JOB based on JOB TITLE
- An index on EMPLOYEE based on EMP LAST NAME

Refined data structure diagram




COVERAGE
50 CALC
EMP-COVERAGE
COVERAGE-NHC
FK (PLAN CODE
EMP ID)
COVERAGE-HC
FK (PLAN CODE
EMP ID)
COVERAGE-dc
FK (PLAN CODE
EMP ID)
NON-HOSPITAL CLAIM
304 CLUSTERED
COVERAGE-NHC
INS-DEMO-REGION
HOSPITAL CLAIM
310 CALC
HOSPITAL CLAIM ID U
DENTAL CLAIM
184 CALC
DENTAL CLAIM ID U
NHCLAIM-PROC
FK (NON-HOSP CLAIM ID)
NON-HOSP-NDX
ASC (NON-HOSP CLAIM ID
PROCEDURE NUMBER) U
DCLAIM-PROC
FK (DENTAL CLAIM ID)
PROC-NDX
ASC (DENTAL CLAIM ID)
PROCEDURE NUMBER) U
NON-HOSP PROCEDURE
85 CLUSTERED
NHCLAIM-PROC
DENTAL PROCEDURE
85 CLUSTERED
DCLAIM-PROC

Chapter 13: Choosing Physical Tuning Options

This section contains the following topics:

Overview

Physical tuning options

The following database options can be used to ensure optimal performance in individual business transactions:

- **Placement of entities in areas**—To facilitate certain processing operations, you can instruct CA IDMS/DB to divide the database into separate areas. Each area can contain one or more entities.

  You can also sometimes simplify application processing, recovery procedures, and unload/load operations by segmenting the database.

- **ata compression**—To save disk space, you can instruct the database to compress data before it is stored and decompress it when it is retrieved.
- **Relationships and tuning options**—When relating entities, you can establish linked or unlinked relationships. Linked relationships can be used to optimize performance in applications that process related entities.
- **Index key compression**—To save disk space, you can instruct CA IDMS/DB to compress indexes.
- **Non-SQL tuning options**
  - **Multimember relationships**—A single relationship is maintained for multiple child entity types.
  - **Direct location mode**—You can assign this location mode to an entity when the application programmer must be able to explicitly specify the physical location of entity occurrences in the database.
  - **Variable-length entities**—You can collapse two entities involved in a one-to-many relationship where the many entity can contain a variable number of occurrences.
  - **Database procedures**—You can write and compile database procedures to be executed at application runtime when a program accesses an area or entity to perform predefined programming functions such as data compression and decompression.
  - **CALC duplicates options**—You can specify options for nonunique CALC keys specifying how these nonunique occurrences will be stored in the database.
  - **Relationship tuning options**—You can specify options as part of the definition of a relationship to specify the order of child occurrences, how the occurrences will be linked with each other, how new occurrences are introduced into the relationship, and how existing occurrences can be modified.
  - **Index tuning options**—You can specify options as part of the definition of an index to provide for unlinking the index and for determining the order in which entity occurrences will be referenced in the index, how new occurrences are introduced into the index, and how existing occurrences can be modified.

Each of these tuning options is described in detail below.

Placement of Entities in Areas

Why separate entities?

To facilitate certain processing operations, you can instruct CA IDMS/DB to divide the database into separate areas. Each area can contain one or more entities. You place database entities in separate areas to:

- **Minimize processing interruptions** that might be caused by backup and recovery procedures. CA IDMS/DB provides standard system utility programs that allow the system operator to rollforward/rollback or dump/restore only those areas in a database that require backup and recovery. Before performing backup and recovery procedures, the operator typically varies each area or file that is currently held in update usage mode to retrieval (or offline mode). Once an area has been varied to retrieval or offline mode, further update processing is not allowed. By assigning entities to separate areas, you can ensure that backup and recovery procedures impact the minimal number of applications.

  For further information on backup and recovery, see *CA IDMS Utilities Guide* and *CA IDMS Database Administration Guide*.

- **Reduce time required to perform maintenance activities** (such as unload and reload by area). By separating entities into separate areas, you make the amount of data processed smaller, which, in turn, reduces the time required for the processing.

- **Reduce cluster overflow.** The impact of large cluster sizes can be reduced by separating one or more entity types into separate areas. This is especially effective if less-frequently accessed entities are separated.

- **Improve efficiency of serial processing**. If an entity (or entities) is to be retrieved mainly by area sweeps, that entity (or entities) should be assigned to a separate area.

Guidelines

Consider the following general guidelines for assigning entities to database areas:

- Whenever possible, place indexes in separate areas. If two or more indexes can be accommodated by the same page size, you can place the indexes in the same area. If using a non-SQL implementation, consider segregating each index in its own page range if they are in the same area or if the indexes are restricted to separate page ranges.

- In general, you should store only one type of entity cluster in each area of a database.

- Nonclustered entities can be placed together in a separate area or can be included in an area containing a cluster, provided that CALC overflow will not be a problem.

Segmentation of Databases

By segmenting the database, you can simplify application processing, recovery procedures, and unload/load operations. CA IDMS/DB allows you to create databases that are segmented according to:

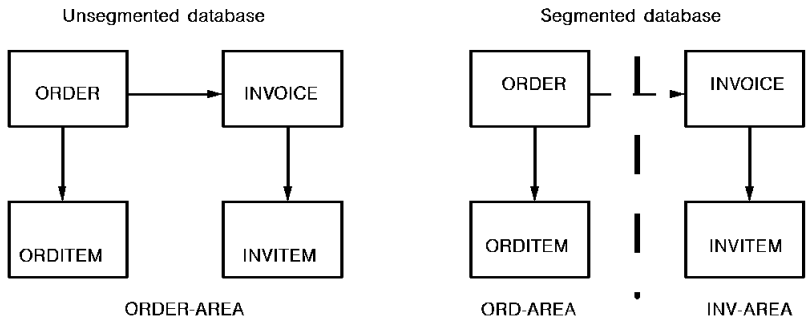
- Groups of entities
- Logical keys

Segmenting by Groups of Entities

To facilitate processing of the same data by different application programs, you can create a database that is segmented by groups of entities, as shown below.

To create such a database, you assign entities to separate database areas and use only unlinked (as opposed to linked) relationships between entities in different areas. See "Linked and Unlinked Relationships" later in this chapter for further information on types of relationships.

Database segmented by groups of entities



Advantages

A database segmented by entity is advantageous because it:

- Eliminates the need to perform maintenance for linked relationships that cross areas and facilitates and shortens unload/reload operations.
- Allows certain application programs to remain active while parts of the database are being recovered or restructured.

Considerations

Although a database segmented by entity can facilitate certain processing functions, it can sometimes complicate processing of child entities. If an application requires the ability to group child entities by parent, the DBMS must use additional system resources to access the related entities that are stored in different areas.

Segmenting by Logical Key

Segmenting by logical key is used to separate a large non-SQL-defined database into identical segments based on the value of one or more data elements. For example, you might separate employee data by company code, each company within Commonweather Corporation having its own segment of the database.

Note: The key field on which the segmentation is performed may or may not actually exist as a data element in some entity of the database.

Segmenting by key value in a non-SQL implementation

To segment by key value in a non-SQL implementation:

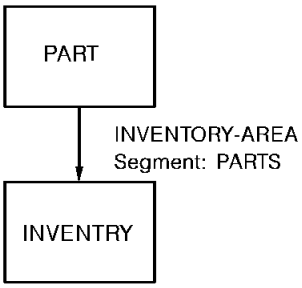
1. Define a single schema that describes the database.
2. Define a set of subschemas associated with the schema.
3. Define a segment for each physical implementation of the database. Each segment must contain the same named set of areas. Use separate page ranges or page groups to distinguish each segment.
4. If necessary, define a database name for each segment, including the corresponding segment and additional segments for other data accessed by the application.
5. Provide a mechanism to direct each application program to the correct segment by specifying the DBNAME or segment name on its BIND RUNUNIT statement.

Segmenting by key value in an SQL implementation

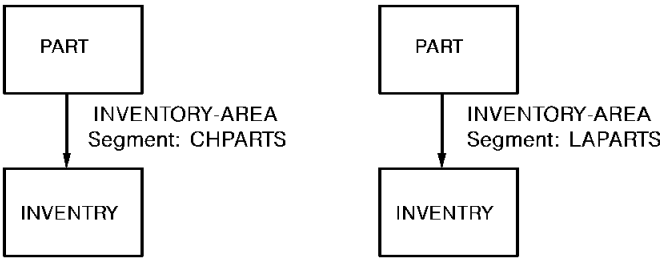
To segment by key value in an SQL implementation:

1. Define a segment for each logical division of the database. Each segment must contain the same named set of areas.
2. Define a schema for each logical division. Each schema will describe tables in one of the segments.
3. Define the identical set of tables in each schema.
4. For each application, create a set of access modules, one for each schema.
5. Provide a mechanism to direct processing to the correct access module at runtime.

Database Implementation by Key Value



*Although the company's Chicago and Los Angeles warehouses maintain separate inventories, they share a common database due to common processing.



*The company's Chicago and Los Angeles warehouses maintain separate databases and use database name tables to direct the programs to the correct area(s).

Advantages

A database implementation by key value is advantageous because it:

- Simplifies recovery operations by permitting certain application programs to remain active while parts of the database are being recovered or updated.
- Facilitates and shortens unload/load operations.
- Allows for distribution of an organization's processing to multiple machines and sites.

Considerations

While a database that is implemented by key value facilitates certain processing functions, it complicates simultaneous processing of all segments.

In an SQL environment, you could create a view of all the tables at once to access all segments at one time.

In a non-SQL environment, you would have to bind concurrent run units to access all segments at one time. An alternative is to bind rununits serially.

Data Compression

Conserving disk space

To conserve disk space, you can instruct the database to compress data before storage and decompress it after retrieval. There are three ways to compress and decompress data:

- CA IDMS Presspack
- IDMSCOMP and IDMSDCOM database procedures
- User-written procedure

These procedures are invoked automatically by the DBMS as data is stored and retrieved.

Note: Only CA IDMS Presspack is available for SQL-defined data.

Advantages and disadvantages of data compression

The following table summarizes the advantages and disadvantages of data compression.

Efficiency Considerations	Potential Impact
I/O	By compressing an entity, you conserve storage resources, allowing the system to fit more entities on each database page. If you can fit all entity occurrences associated through a particular relationship on a single page, the system will only perform one I/O to access these entities.
CPU time	Compressing data requires some extra CPU time to perform compression/decompression processing.
Space management	Compression can be used to conserve considerable amounts of storage.
Contention	No difference.

Considerations for using CA IDMS Presspack

CA IDMS Presspack uses Huffman techniques to compress database entities. The techniques include assigning unique bit string codes of different lengths to single character and character strings. These codes substitute for the character and character strings in the entities.

To assign the codes, CA IDMS Presspack uses character and character-string frequencies of occurrence. It assigns shorter codes to the most frequently occurring characters and character strings. To those that occur less frequently, CA IDMS Presspack assigns longer codes.

CA IDMS Presspack compresses both textual and nontextual data.

For further information about CA IDMS Presspack, see *CA IDMS Database Administration Guide* and *CA IDMS Presspack User Guide*.

Considerations for using IDMSCOMP and IDMSDCOM

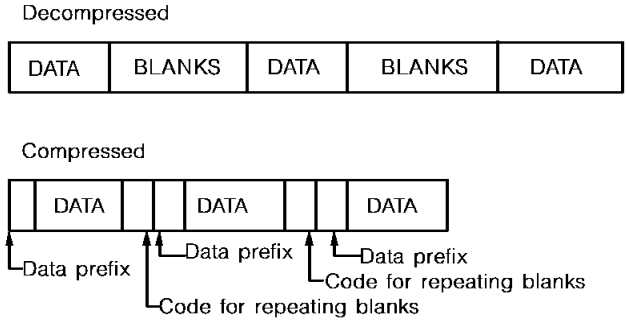
IDMSCOMP and IDMSDCOM are supplied with CA IDMS/DB. They are placed in the load (core-image) library at installation time and are also provided in source form so you can modify them if necessary. You can also write your own database procedure or use other commercially available compression/decompression procedures.

For further information about database procedures, see *CA IDMS Database Administration Guide*.

To compress data, IDMSCOMP performs the following conversion procedures:

- 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 any of a number of commonly used character pairs into a 1-byte code.

Data that does not fall into any of the above categories remains unchanged. Each group of unchanged data is prefixed by a 2-byte code. The following diagram shows the compression of contiguous blanks in an entity.



Considerations for user-written procedures

If writing your own compression procedures, you must follow conventions for writing database procedures.

For information on database procedures, see *CA IDMS Database Administration Guide*.

Guidelines for compression

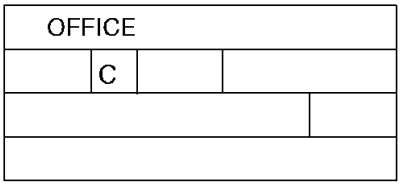
Consider the following guidelines when deciding whether data should be compressed:

- When determining whether or not to compress/decompress an entity, you should consider whether the disk space saved justifies the CPU overhead incurred by the routines.
- The control portion of an entity is not compressible.

  The control portion of an entity includes all data elements up to the last key (CALC, sort, index). Since this portion of an entity is not compressible, it may mean that not enough compressible data exists to justify compression.
- Use compression/decompression procedures for entities that are not updated often. While the compression procedures save considerable disk space, it uses additional CPU time to perform its processing.
- Do not compress entities that start with large groups of repeating characters but lose them over time.
- IDMSCOMP/IDMSDCOM considerations:
  - IDMSCOMP and IDMSDCOM compression procedures operate most efficiently for entities whose occurrences usually contain sizable portions of blanks or binary zeros.
  - Don't use this compression for entities containing only small scattered groups of repeating characters.
  - Data that is stored in packed decimal format is not a good candidate for data compression.

Storage mode

If you decide to compress data in an entity, you should add a storage mode of **C** for the entity on the data structure diagram.



Relationship Tuning Options

What is a relationship?

Entity occurrences are related to one another if the foreign key in a child occurrence has the same value as the primary key in a parent occurrence. You identified relationships in the logical database design process.

Linked and Unlinked Relationships

Linked and unlinked

When implementing these relationships, there are a number of physical tuning options from which to choose. You have already decided whether a relationship is a clustering relationship or not. You must now decide whether to define the relationship as linked or unlinked.

- A **linked** relationship is one in which related entity occurrences are linked to one another through embedded pointers.
- An **unlinked** relationship is one in which no embedded pointers are used to link related entity occurrences.

Advantages of linked relationships

Linked relationships have the following advantages:

- Since there is direct linkage between parent and related child occurrences, linked relationships provide the most efficient means (in terms of CPU and I/O) of retrieving related entity occurrences.
- Unlinked relationships require that a CALC key or index be defined on the foreign key of the child entity.
- An index adds both CPU and I/O to retrieve data and maintain the index. It also requires additional storage space.
- Defining a CALC key on the foreign key is almost as effective as using a linked relationship provided that it does not cause CALC overflow conditions, which increases I/O, CPU, and contention. However, you can define only one CALC key per entity, so that an entity participating as a child in more than one relationship must use indexes for all but one unlinked relationship.
- Linked relationships provide an ordering option that can reduce the need for additional indexes to enforce unique constraints and avoid sorting of retrieved information.

Considerations

Keep the following considerations in mind when using linked and unlinked relationships.

- Self-referencing relationships must always be unlinked.
- Linked relationships require physical restructuring of entity occurrences to add or remove relationships.
- The time required for and impact of maintenance operations, such as unload/reload, can be reduced if relationships between entities in different areas are unlinked. This is particularly important in designing large databases.

Non-SQL considerations

In a non-SQL environment:

- There is no integrity enforcement by the DBMS with an unlinked relationship. Integrity must be enforced by applications or logical record facility path logic.
- There is no relationship clustering with an unlinked relationship. You must use CALC clustering to achieve results similar to clustering.

Note: If CALC clustering results in long CALC chains, CPU, I/Os, and contention might all increase.

You can eliminate foreign keys from child entities if the relationship is linked. This has the following results:

- It reduces storage requirements
- It eliminates the need to update each child occurrence if the parent's key is changed.

  For example, if you change the value of DEPT ID in a department, related employees do not need to be updated.

If you choose to retain the embedded foreign keys, you:

- Have full update SQL access to the data
- Will reduce the number of I/Os required to retrieve foreign key values for nonclustered entities (for example, to retrieve the department ID of an employee)

Unlinked Relationship Tuning Options

In designing an unlinked relationship, define the following:

- Index or calc key on the foreign key of the relationship

Additional Columns in the Foreign Key Indexes

In designing an unlinked relationship, you must define an index or calc key on the foreign key of the relationship. If you use an index to enforce the integrity of the referential constraint, it must contain the columns that make up the foreign key but can contain additional columns. Defining additional columns after the foreign key columns has the potential for reducing disk space requirements and improving performance.

The ability to extend foreign key indexes with additional columns may enable one index to be used for multiple purposes. For example, a table's primary key is often a concatenation of one of its foreign keys with additional columns that together form a unique identifier for each row of the table. A single index can be used to enforce both the integrity of the referential constraint and the uniqueness of the primary key. By eliminating a second index you reduce disk space requirements and the overhead associated with index maintenance.

Including extra columns in a foreign key index may also improve access efficiency by enabling the use of more index scans to identify rows matching selection criteria. The use of an index scan can significantly reduce the number of I/Os needed to satisfy a query.

Defining additional columns in the index key

To define additional columns in the index key, define an index so that the foreign key columns precede any additional columns in the index key. The order of the foreign key columns in the index key must match the order of the referenced columns in some unique index or CALC key on the referenced table.

Linked Relationship Tuning Options

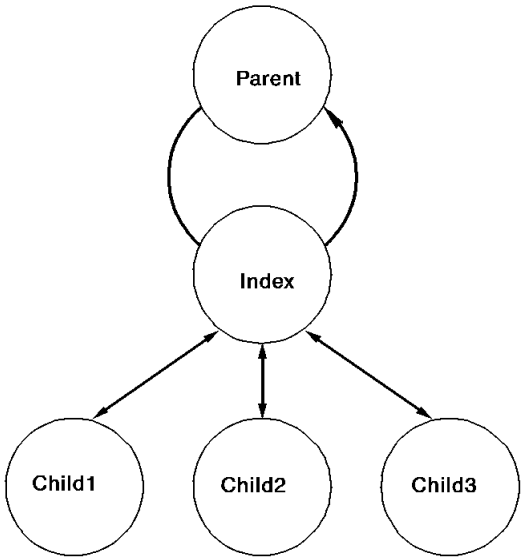
In designing a linked relationship, you specify the following options:

- Type of linkage (chained or indexed)
- Relationship ordering (sorted or unsorted)
- Sort options (order and uniqueness)

Type of Linkage

CA IDMS/DB supports the following types of linked relationships:

- **Chained**—The DBMS maintains relationships based on internal information stored in the prefix of each entity occurrence. This information in the prefix contains the db-key of the logically next occurrence in the relationship.
- **Indexed**—The DBMS maintains relationships through an **index** between a parent and related child occurrences. The bottom level of the index contains the db-keys of the related child occurrences. Each child occurrence contains an **index pointer** that points to the bottom level of the index.



Guidelines

As a general rule, use **indexed** for nonclustered relationships and **chained** for clustered relationships.

An **indexed** nonclustered relationship requires fewer I/Os to add or remove an entity occurrence than a chained nonclustered relationship. This is because the adjacent entity occurrences are not updated; only the index structure needs to be updated. In addition, fewer I/Os are required to retrieve a child occurrence by key in a nonclustered relationship if it is indexed rather than chained.

A **chained** relationship, on the other hand, requires less CPU overhead for maintenance and retrieval than an indexed relationship. It also requires less storage space because there is no index structure. For these reasons, it is a better choice than indexed for clustered relationships because I/Os are not generally a concern.

Note: For databases implemented with SQL, all linked clustered relationships are chained and all linked nonclustered relationships are indexed.

For further information on the structure of indexed relationships, see Chapter 15, "Determining the Size of the Database".

For further information on indexed relationships, see *CA IDMS Database Administration Guide*.

A comparison of indexed and chained relationships

The following table presents a comparison of indexed relationships and chained relationships.

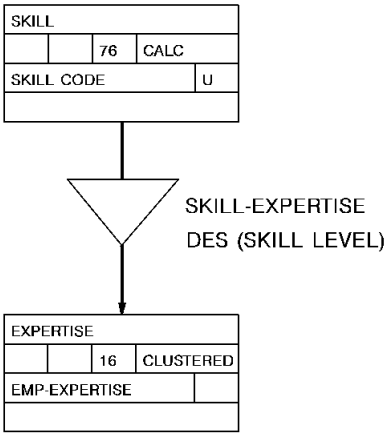
Efficiency Considerations	Potential Impact
I/O	Indexed relationships often require fewer I/O operations to access child entities in nonclustered relationships, especially if the relationship is sorted.
CPU time	Chained relationships use less CPU time for processing of child entities than indexed relationships.
Space management	Chained relationships require less storage space than indexed relationships.
Contention	No difference.

Representing an indexed relationship

To represent an indexed relationship:

- Name the relationship.
- Specify whether the order is **ASC**ending or **DES**cending for each key.
- Identify the data element name(s) to be indexed.
- Specify whether duplicate indexed items are allowed (blank) or not allowed (**U** for unique).
- Specify whether the index key is to be compressed.

The following diagram shows the standard CA IDMS/DB notation for an indexed relationship. The index allows the DBMS to access all EXPERTISE occurrences associated with a particular skill based on skill level in descending order.



Sorted and Unsorted Relationships

You can specify the logical order of child occurrences within each linked relationship:

- **Sorted**— A new entity occurrence is positioned according to the value of one or more of its data elements (called a sort key) relative to the values of the same data elements in other related child occurrences.
- **Unsorted**— A new entity occurrence is positioned according to a predefined order within the relationship.

  For example, all new entity occurrences might be positioned ahead of all existing occurrences.

Advantages of a sorted relationship

Through a sorted relationship:

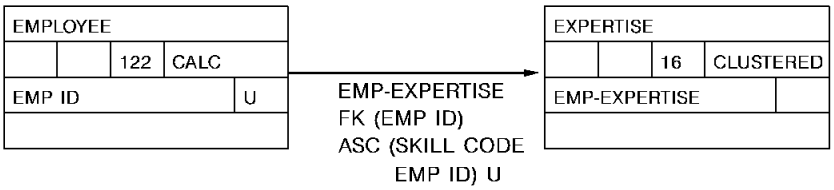
- A program can retrieve a child occurrence directly by key, thus reducing CPU.
- A program can retrieve child occurrences data in order, thus avoiding sorts.
- Unique constraints can be enforced without the need for additional indexes.

Considerations for sorted relationships

Maintaining the relationship's order during update operations requires increased CPU and a greater number of I/Os than an unsorted relationship.

Enforcing unique constraints

Sorted relationships can be used to enforce unique constraints as an alternative to a CALC key or index. For example, you can eliminate the EXP-NDX index in the Commonweather Corporation by defining either the SKILL-EXPERTISE or the EMP-EXPERTISE relationship as a unique sorted relationship.



To eliminate the index, you must either:

- Define SKILL-EXPERTISE as sorted on EMP ID with the unique option.

  or

- Define EMP-EXPERTISE as sorted on SKILL CODE with the unique option

Either approach ensures that no employee is assigned duplicate skills.

Sorted order

You can choose to sort in **ascending, descending,** or **mixed** order.

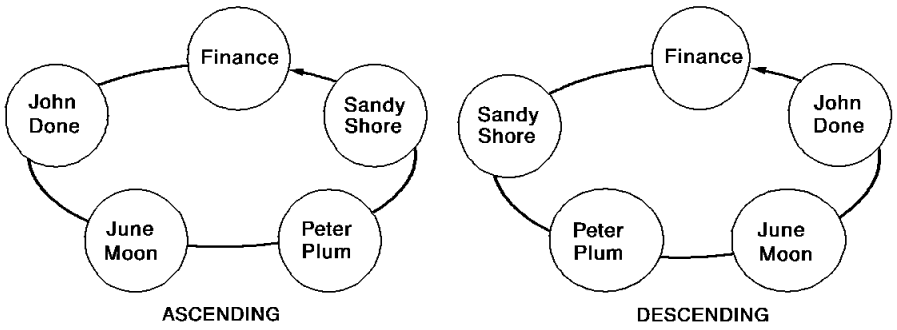
As a general rule, choose the sort order to reflect the most commonly desired retrieval order. However, the sequence chosen for a chained relationship can have an impact on performance in update transactions. This will allow the DBMS to locate the point of insertion more quickly.

If new entity occurrences typically have sort key values greater than existing occurrences, the relationship should have a descending sort order. Conversely, if new occurrences have sort keys lower than existing occurrences, ascending is preferable.

For example, new occurrences of dated entities are usually stored with higher dates than previously stored occurrences. If this is the case, you should specify descending for a chained relationship sorted by date.

More Information

For more information concerning the usage of numeric fields as part of a sort key, see Zoned and Packed Decimal Fields as IDMS Keys (see page 301).



Nonsorted Order

If the entity occurrences in the relationship are not to be sorted, you can specify the logical order of child entity occurrences within each occurrence of a relationship. You determine how a new child is placed in a relationship by specifying one of the following orders:

- **FIRST** creates a LIFO (last in, first out) order. The new entity is positioned at the beginning of the relationship.
- **LAST** creates a FIFO (first in, first out) order. The new entity is positioned at the end of the relationship.
- **NEXT** creates a simple list. The new entity is positioned immediately after the current (most recently accessed) entity. The NEXT order is recommended as a default.
- **PRIOR** creates a reverse list. The new entity is positioned immediately before the current entity.

Flexibility

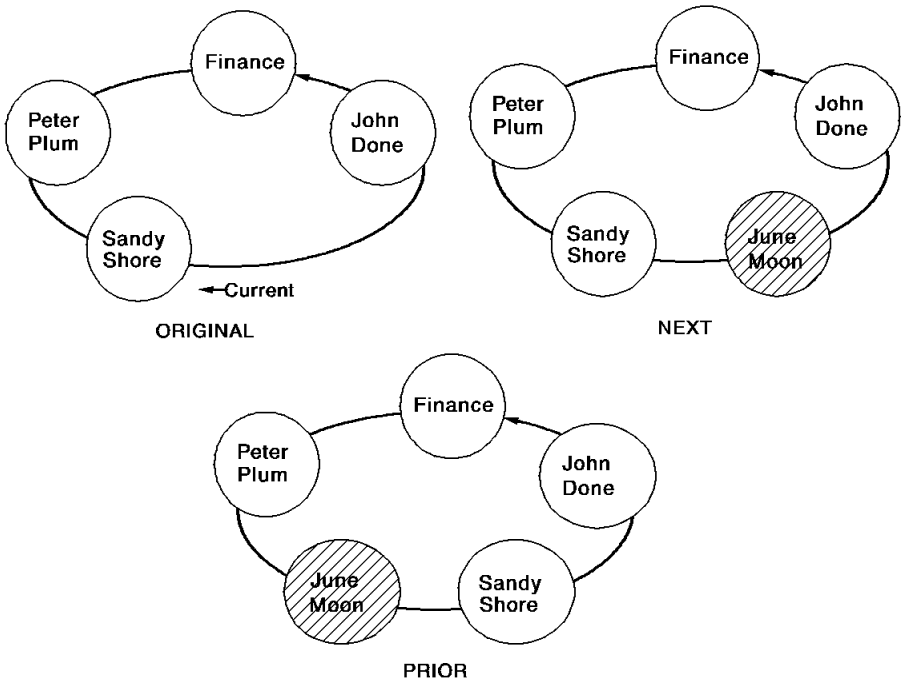
The NEXT and PRIOR orders provide more flexibility than the FIRST and LAST options; the programmer can connect an entity anywhere within the relationship by establishing currency before or after the point of insertion. When the FIRST and LAST options are assigned, the programmer can be certain of the positioning of new entities, regardless of set currency.

Note: The PRIOR and LAST options require prior pointers.

For more information on pointers, see "Linkage" later in this chapter.

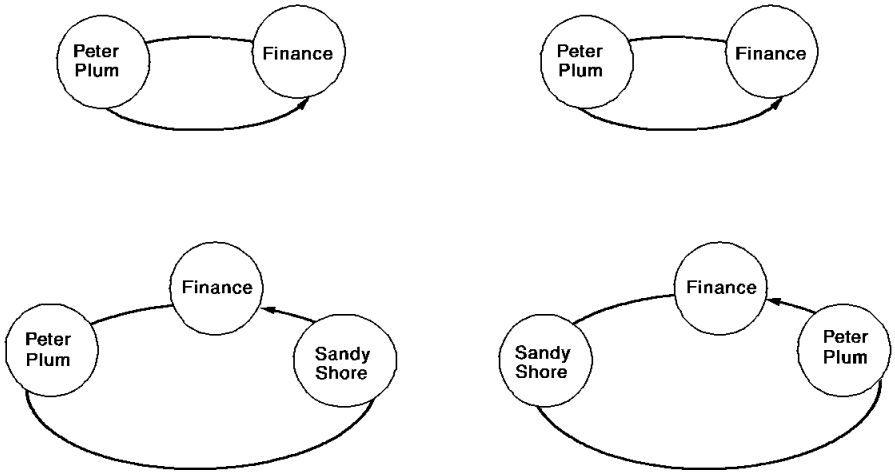
Next and prior order example

In the example below, assume that a program is positioned on SANDY SHORE before it stores JUNE MOON in the database. In a relationship defined with the NEXT order, JUNE MOON will be stored after SANDY SHORE. In a relationship defined with the PRIOR order, JUNE MOON will be stored before SANDY SHORE.



First and last order example

Suppose two entities are added in the following order: PETER PLUM, then SANDY SHORE. In a relationship defined with the FIRST order, the entity stored most recently (SANDY SHORE) will be returned first. In a relationship defined with the LAST order, the entity stored first (PETER PLUM) will be returned first.



Additional Sort Options

Standard and natural collating sequence

You can specify either of two collating sequences for sorted relationships:

- **Standard** collating sequence for sorted relationships orders key fields based on their EBCDIC collating sequence without regard to data type.
- **Natural** collating sequence for sorted relationships orders key fields based on their data type. This means that negative numeric values will collate lower than positive values.

In the example below, assume that the values are packed or zoned decimal numbers. They are ordered first using the natural collating sequence and then using the standard collating sequence.

Natural	Standard
-4268.50	15.26
-351.78	144.83
-258.00	-258.00
15.26	-351.78
144.83	2594.38
2594.38	-4268.50

Duplicates options

You can specify options for relationships indicating how nonunique occurrences will be logically placed in a sorted relationship. You can specify **duplicates first** or **duplicates last**

- **Duplicates first**— The duplicate entity occurrence will be logically placed in the relationship *before* the entity occurrence already having that sort key.
- **Duplicates last**— The duplicate entity occurrence will be logically placed in the relationship *after* the entity occurrence already having that sort key.

Duplicates not allowed in the non-SQL definition is equivalent to unique.

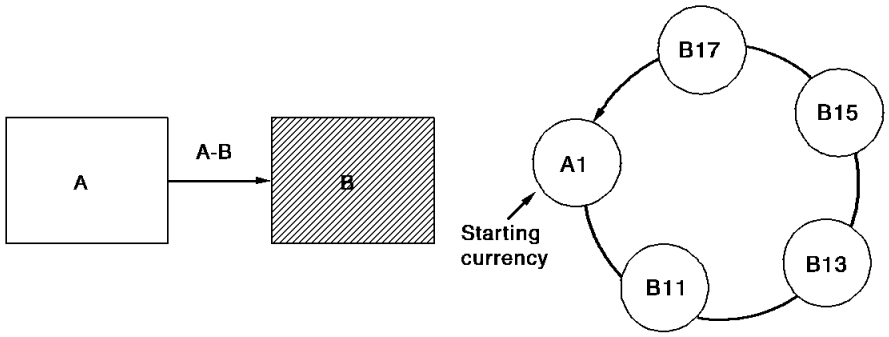
A relationship can be sorted in either ascending or descending order. The duplicates option for a sorted relationship determines what happens when a user tries to store an entity with a duplicate sort key value.

You can order the sorted relationship entity occurrences with duplicate key values as **duplicates first**, **duplicates last**, as discussed above, or in child db-key sequence. This option speeds retrieval by reducing I/O.

Use sorted relationships to simplify programming

Sorted relationships simplify programming effort by allowing the programmer to specify a symbolic key value for storage, retrieval, and positioning of an entity occurrence in the database. By using sorted relationships, the programmer need issue only one DML statement to locate an entity in the database. To locate an entity in a FIRST, LAST, NEXT, or PRIOR relationship, the programmer must walk the relationship by issuing several DML statements.

The diagram below shows the use of sorted relationships to simplify programming.



Unsorted

```
0200-GETREC.
  OBTAIN NEXT B WITHIN A-B.
  IF DB-END-OF-SET
     THEN GO TO 0900-NOREC.
  PERFORM IDMS-STATUS.
  IF B-KEY NOT = 'B15'
     THEN GO TO 0200-GETREC.
```

Sorted

```
0200-GETREC.
  MOVE 'B15' TO B-KEY.
  OBTAIN B WITHIN A-B USING B-KEY.
  IF DB-REC-NOT-FOUND
     THEN GO TO 0900-NOREC.
  PERFORM IDMS-STATUS.
```

Use sorted relationships to enhance online or batch processing

Since sort routines incur considerable CPU overhead, they are rarely used in online programs. Sorted relationships are therefore useful for sequencing data for online display. They are also useful in the batch environment: a batch program can process sorted input transactions very efficiently in sorted relationships.

Linkage

Each entity in the database carries one, two, or three pointers for each *chained* relationship in which it participates. You should usually include all allowable pointers for each entity:

- **Next pointer**—Required for all relationships in which the entity participates as parent or child; the next pointer is the database key of the next entity in the relationship. The last child entity in a relationship points to the parent.
- **Prior pointer**—Optional for all relationships in which the entity participates as parent or child; the prior pointer is the database key of the prior entity in the relationship. The first child entity occurrence in a relationship points to the parent.
- **Owner pointer**—Optional for all relationships in which the entity participates as a child; the parent pointer is the database key of the parent entity occurrence.

Omitting prior pointers

Prior pointers can be omitted under the following conditions:

- Child entity occurrences in the relationship will not be erased or disconnected except by walking the set.
- Child entity participates as a child in no other relationship.
- Order is not LAST or PRIOR (see "Nonsorted Order" above).
- The FIND/OBTAIN LAST or FIND/OBTAIN PRIOR DML functions will not be used for the relationship.

Omitting owner pointers

Owner pointers (db-keys pointing to the parent) can be omitted under the following conditions:

- The parent will not be accessed from a child occurrence.
- The FIND/OBTAIN OWNER DML function will not be used for the relationship.

Note: Be sure to include an OWNER pointer for any entity that participates as a child in more than one relationship since the child entity is probably an entity created to implement a many-to-many relationship. In this case, the system will most likely need to access parent entities from the child entities regularly.

Pointers in indexed relationships

The parent of an *indexed* relationship has the following mandatory pointers:

- **Next pointer**— Points to the first occurrence of an SR8 entity (an internal entity used to hold the index)
- **Prior pointer**— Points to the last occurrence of an SR8 entity

For further information on the structure of an index, see Chapter 15, "Determining the Size of the Database".

The child entity occurrence of an indexed relationship has one mandatory and one optional pointer:

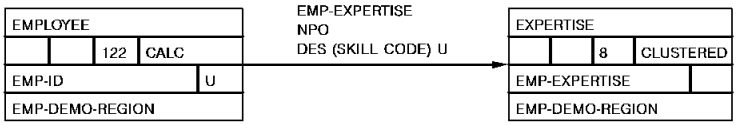
- **Index pointer**—This pointer is required; it is used to access the SR8 entity that owns a particular child entity occurrence.
- **Owner pointer**—This pointer is optional; it points to the parent of the relationship

For further information on the structure of indexed relationships, see Chapter 15, "Determining the Size of the Database".

For further information on indexed relationships, see *CA IDMS Database Administration Guide*.

Representing linkage

Represent relationship linkage on the data structure diagram by identifying the pointers to be used. For example, specifying **NPO** indicates that next, prior, and owner pointers are to be used.



For an indexed relationship, specify **I** or **IO**.

Membership Options

Membership options determine how an entity is connected to and disconnected from a relationship. These options affect the use of the DML STORE, CONNECT, DISCONNECT, and ERASE statements.

You define membership options in two parts. The first part indicates the manner (**mandatory** or **optional**) in which the entity is disconnected from a linked relationship. The second part indicates the manner (**automatic** or **manual**) in which the entity is connected to a linked relationship.

Disconnect options

The disconnect options operate as follows:

- **Mandatory**— A child occurrence cannot be disconnected from the relationship without also being erased from the database (that is, the DML DISCONNECT verb cannot be issued against entities in the relationship).
- **Optional**— A child occurrence can be disconnected from a relationship by the DISCONNECT verb. The entity occurrence remains in the database and is accessible in other ways; it can be connected to another relationship.

The mandatory/optional membership specification affects the outcome of the DML ERASE statement. If any of the ERASE options (PERMANENT, SELECTIVE, ALL) is specified when an ERASE statement is issued against an entity, all mandatory entities owned by that entity are also erased. Optional child entity occurrences are left as is, disconnected, or erased, depending on the ERASE option specified.

Mandatory disconnect

The disconnect option is usually specified as mandatory. However, do not specify the mandatory disconnect option when:

- An application requires the ability to dissociate a child entity occurrence from its parent (usually with the intention of associating the child with another parent occurrence). At Commonweather Corporation, employees sometimes need to be transferred from one department to another. Therefore, the disconnect option for the DEPT-EMPLOYEE relationship must be specified as optional

  Important! Be careful when using the optional disconnect option for child entities of a relationship stored clustered around that relationship. If the entity is later disconnected from its original parent and connected to another, CA IDMS/DB does not physically relocate the entity; for all practical purposes, that entity is no longer clustered around its parent.

- An application requires the ability to erase a parent entity without erasing the child entities (using the ERASE PERMANENT and ERASE SELECTIVE functions). Suppose the Commonweather Corporation decides to close an office in a certain city. In this case, the office should be erased, but the employees who work in that office should not be erased.

Connect options

The connect options operate as follows:

- **Automatic**—The membership of an entity in a relationship is established automatically by the DBMS whenever a child occurrence is stored in the database.
- **Manual**—The membership of an entity in a relationship is not established when a child occurrence is stored. Membership must be established explicitly by using the DML CONNECT statement.

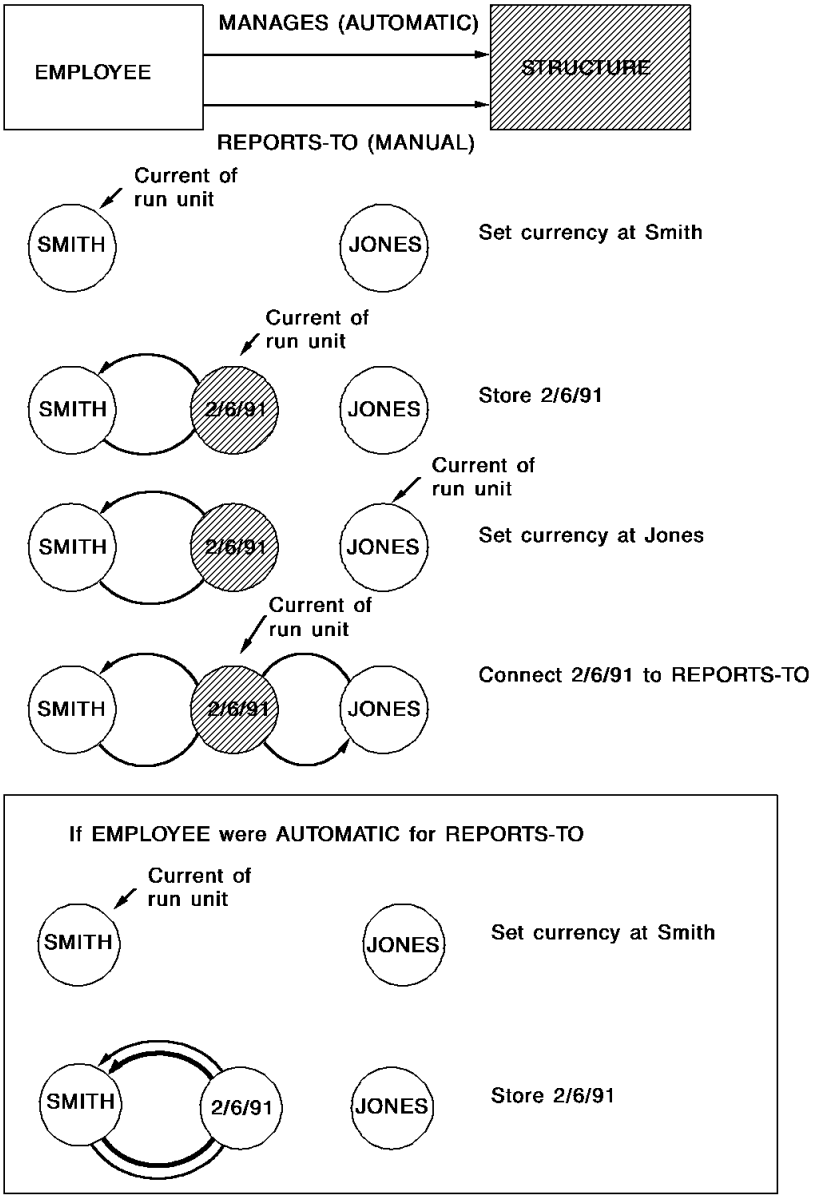
Disconnect and connect options are combined to form membership options:

- **MA**— Mandatory automatic
- **MM**— Mandatory manual
- **OA**— Optional automatic
- **OM**— Optional manual

Automatic connect

The connect option is usually specified as automatic. However, do not specify the automatic connect option when:

- An application requires the ability to store a child entity without associating it with any parent. For example, at Commonweather Corporation, an employee can join the company without first being assigned to a department. Therefore, the manual option must be specified for the DEPT-EMPLOYEE relationship.
- If two relationships exist between the same two entities representing a self-referencing relationship, only one of the relationships can be automatic; the other must be manual. Otherwise, a child would be connected to the same parent occurrence in each relationship, as shown below.

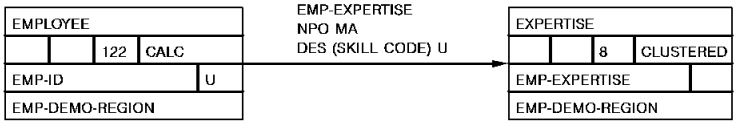


Guidelines

The manual connect and optional disconnect options permit greater flexibility but require more programming effort. Additionally, they provide less control over data integrity. You should therefore choose the **mandatory automatic (MA) membership option**, unless there exists a special business requirement for optional disconnect and/or manual connect functions.

Representing membership options

Represent membership options for a relationship on the data structure diagram by specifying the membership options to be used: MA, MM, OA, or OM.



Removing Foreign Keys

Since all defined relationships in a database implemented with non-SQL are linked, you have the option of removing foreign keys from the child entity. This:

- Reduces storage requirements
- Eliminates the need to update each child occurrence if the parent's key is changed

If you choose to retain the embedded foreign keys, you:

- Have full update SQL access to the data
- Might reduce the number of I/Os required to retrieve foreign key values for nonclustered entities (for example, to retrieve the department ID of an employee)

Index Key Compression

To conserve disk space, you can instruct the database to compress an index key before storage and decompress it after retrieval. The index key is compressed in the same way that data is compressed. (For more information, see "Data Compression" earlier in this chapter.)

Non-SQL Tuning Options

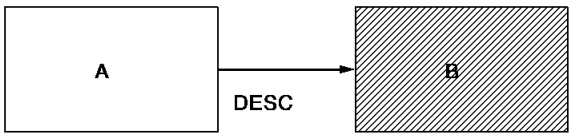
Sorted relationship considerations

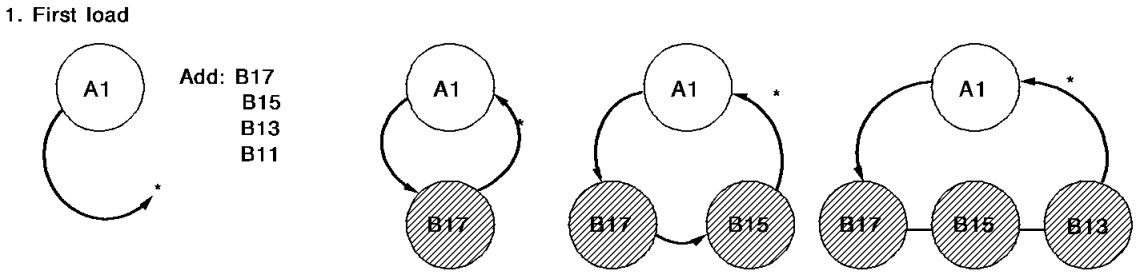
When you store an entity occurrence in a sorted chained relationship, the DBMS searches the relationship in the next direction, starting with the current entity occurrence. If the new occurrence cannot be inserted in the next direction, the DBMS establishes currency on the parent entity occurrence and begins the search from this occurrence (moving in the next direction). When you store an entity occurrence in a sorted indexed relationship, the DBMS searches the occurrences starting from the top of the index structure.

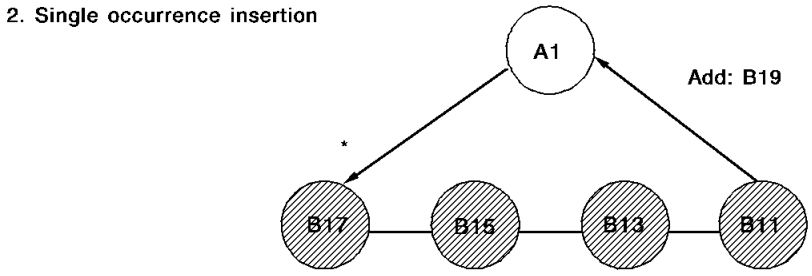
Note: If the DUPLICATES FIRST option is specified for a sorted relationship and the key of the current entity of set is equal to the key of the entity to be stored, the DBMS must begin its search for the insertion point from the owner entity.

Store operations are executed most efficiently when the new entity can be inserted either at the very beginning or the very end of the relationship. If new entities are consistently stored in ascending order, you should perform one of the following procedures to ensure that insertions of new entity occurrences into the relationship will be performed efficiently:

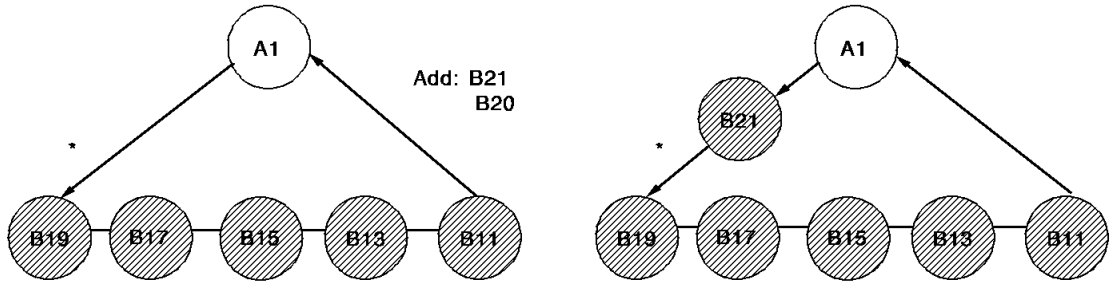
- Assign the **descending** sort sequence to the relationship. In this case, the sequence in which entities are sorted in the relationship is the opposite of the sequence in which new entities are added, as shown below.







3. Multiple insertions



*Insertion point for next record

Note: When you write a program to perform the initial load of the database, plan to sort the entities in the *same* order as the relationship order to optimize processing efficiency. For example, if dated entities are maintained in a relationship that is sorted in descending order, sort the initial load file in descending order before performing the load.

- If you have the option to sort input entities before executing the store operation, you may want to define the sort order as **ascending** and allow the programmer to issue program statements that optimize efficiency. In this case, you should ensure that the programmer establishes currency at the end of the relationship before issuing the store statement command:

  ```
  FIND OWNER
  FIND LAST IN SET
  STORE
  ```

  Remember that you must include PRIOR pointers if you plan to let programmers issue FIND/OBTAIN LAST statements against a chained linked relationship. Without PRIOR pointers, the DBMS must walk the entire linked relationship in the next direction to access the LAST entity.

For more information on pointers, see "Linkage" later in this chapter.

If input entities are consistently stored in descending order, perform one of the following procedures:

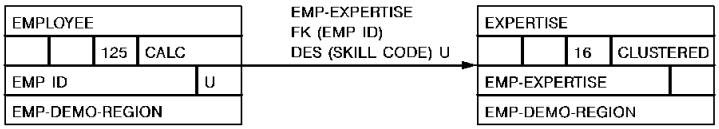
- Assign the ascending sequence to the relationship.
- Have programmers establish currency at the beginning of the relationship before issuing the store command:

  ```
  FIND OWNER
  STORE
  ```

For further information on the DML statements used to access the database, see *CA IDMS DML Reference Guide for COBOL*.

Representing a sorted relationship

Represent a sorted relationship on the data structure diagram by specifying **ASC** or **DES** and the name of the sort key as part of the relationship specification.

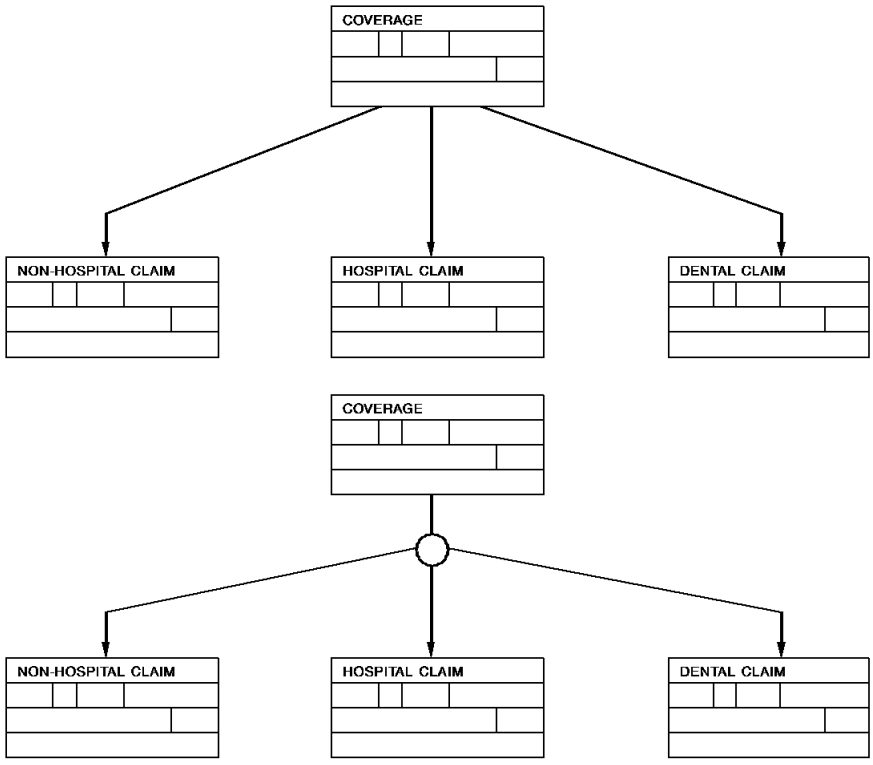


There are additional tuning options available to non-SQL implementations. These are described in this chapter.

Multimember Relationships

What is a multimember relationship?

A multimember relationship is a single relationship maintained for more than one child entity type.



Multimember relationships eliminate the overhead of carrying pointers (db-keys) in the parent entity for additional relationships.

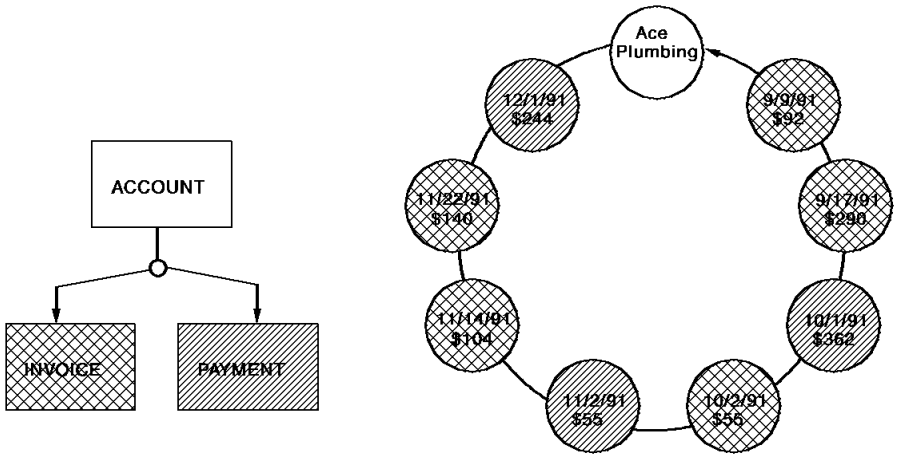
However, to retrieve specific entity occurrences in multimember relationships, the database often must access occurrences of unwanted entity types.

Guidelines

Generally, multimember relationships should be used only when:

- **The different child entity types are usually processed together.**

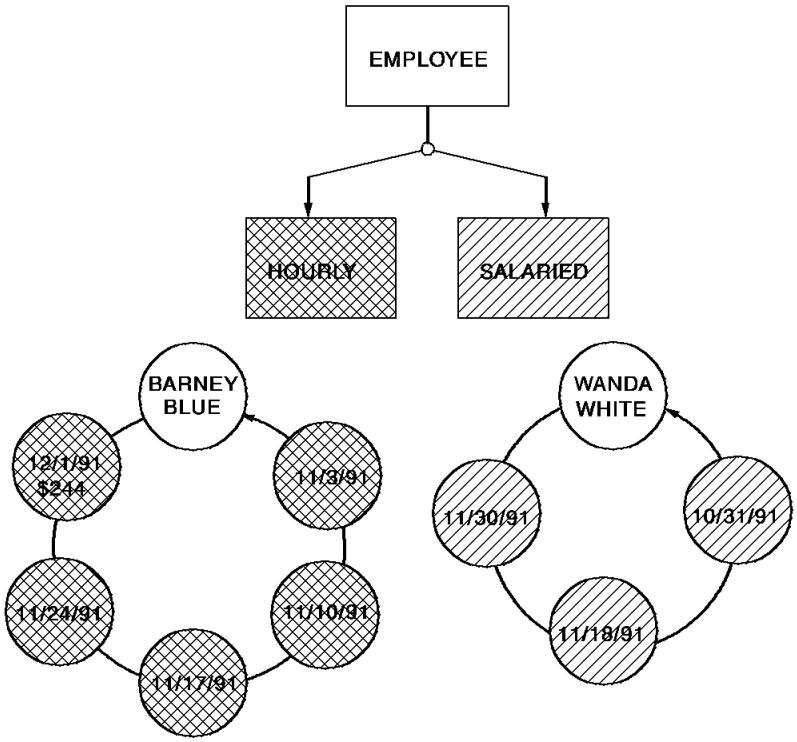
  For example, since the ACCOUNT, INVOICE, and PAYMENT entities are usually processed together, you might want to create a multimember relationship to relate these entity types, as shown below.



Applications that use this accounts receivable structure generate statements that contain details of an account's invoices and payments since the last statement, in order by date. If the INVOICE and PAYMENT entities are maintained in separate relationships, an application program will have to merge them into the proper sequence. If the entities are maintained in one relationship, they are already in order.

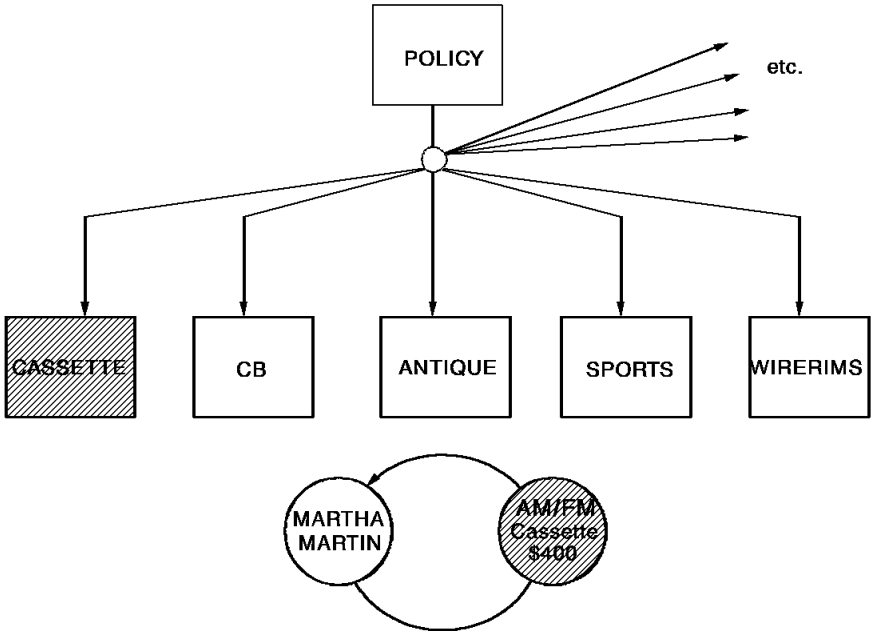
- **The different child entity types are mutually exclusive.**

  Suppose each employee in a corporation is paid on either an hourly or salaried basis. You may want to create a multimember relationship to relate the EMPLOYEE, HOURLY, and SALARIED entities, as shown below.



- **Child entities are of many types, but each child entity type has only a few occurrences.**

  In an auto insurance database, a policy may have many riders, each requiring a different format. However, most policies have no more than a few riders attached. If a relationship were maintained between a policy and each potential rider, the policy entity would require at least five sets of pointers, most of them unused, instead of one, as shown below.



  In all other cases, you should maintain a separate relationship for each entity type.

Considerations

- A multimember relationship cannot be an indexed relationship.
- When accessing a multimember relationship through the logical record facility, only one of the child entity types can be accessed in each pass through the relationship. This means that several passes through a relationship might be necessary to access all child entity types.

  For further information on accessing a multimember relationship through logical records, see the *CA IDMS Logical Record Facility Guide*.
- Multimember relationships should not include both clustered child entity types and nonclustered child entity types. If both types of entities are included in a multimember relationship, I/O performance will be degraded. The system may have to perform additional I/Os to access the clustered child entity occurrences (because the nonclustered child occurrences are distributed throughout the database).

A comparison of multiple relationships and multimember relationships

The following table presents a comparison of multiple relationships and multimember relationships

Efficiency Considerations	Potential Impact
I/O	No difference.
CPU time	Multimember relationships may require more CPU time to process related entities than multiple relationships.
Space management	Multimember relationships eliminate the overhead of carrying pointers in the parent entity for extra relationships.
Contention	In some situations, multimember relationships may cause more entity contention than multiple relationships. If an entity that participates in a multimember relationship is updated often, locking of a modified occurrence of this entity by one transaction may prevent other transactions from accessing occurrences of other entities in the relationship. Therefore you may want to create a separate relationship for a frequently updated entity.

Direct Location Mode

In rare situations, the application program has to have control over an entity's placement in the database. If the application programmer must be able to identify explicitly the location of entity occurrences in the database, you should assign the direct location mode to the entity type. This location mode provides programmers with rapid access to database entities and allows them to control the clustering of entities.

Store entities chronologically

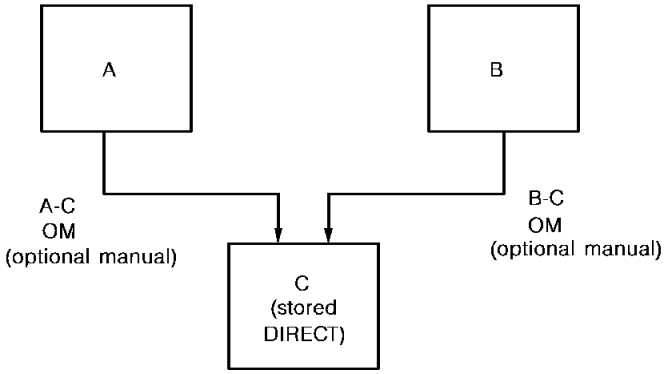
Use direct location mode to store entities chronologically. The direct location mode can be used to arrange entity occurrences serially in a database area. The programmer can arrange entities serially by instructing CA IDMS/DB to store each entity on the same page as the preceding entity. CA IDMS/DB either stores the entity on the same page or on the next page(s), as space availability permits.

Ensure effective clustering

Use the direct location mode to ensure effective clustering. If a child entity has two different parent entities, you may want to take responsibility for clustering occurrences of the child entity. Suppose occurrences of entity C are related to an occurrence of entity A in some instances and by an occurrence of entity B in other instances. You would need to be able to cluster each occurrence of C with its appropriate parent entity (an occurrence of either A or B).

You can achieve effective clustering in this situation by assigning the direct location mode to entity C and the OM (optional manual) membership option to both relationships. Whenever a C entity occurrence must be stored in the database, the application programmer can then connect the entity to its appropriate relationship and cluster the entity with its parent.

For more information on membership options, see "Membership Options" later in this chapter.



However, you should also plan on writing your own unload and reload program for the C entity, since the DBMS does not know how to locate C entities.

Considerations

If the direct location mode is chosen, the entity should either be a child in a relationship or have an index defined on it. If neither of these is true, the only method to access an occurrence is through an area sweep. In most cases, clustering around an index or a relationship is a better storage strategy.

Representing the direct location mode

Represent the direct location mode on the data structure diagram by specifying DIRECT for the location mode. Do not name a CALC key or a relationship.

Variable-Length Entities

Use a repeating element in a variable-length entity instead of two separate entity types when:

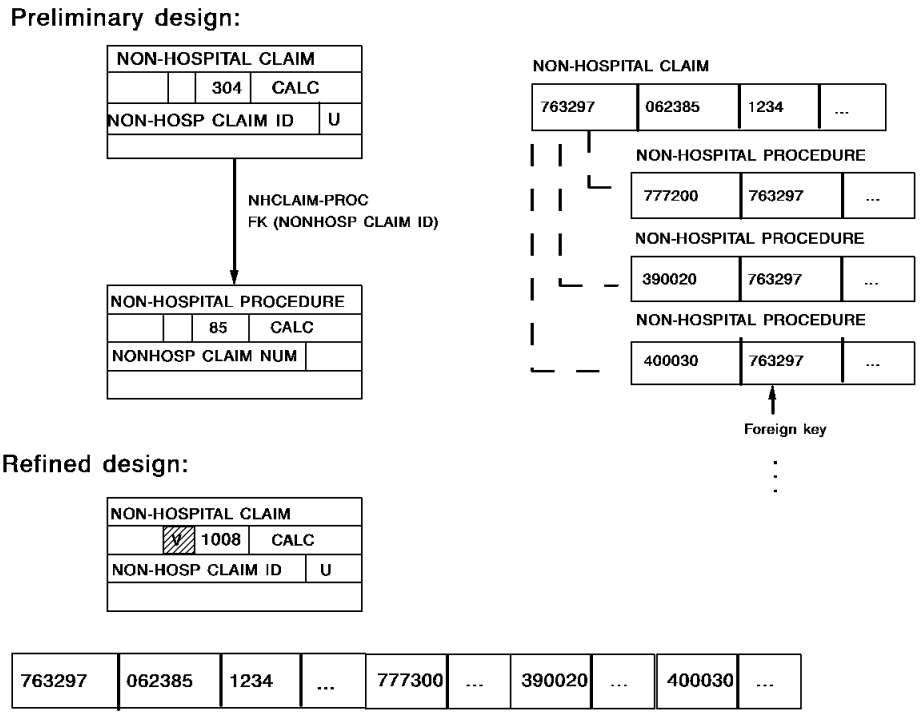
- **The "many" portion of the relationship does not participate in other relationships.** Once you have created a repeating data element, you cannot relate the data in this element to other entity types.
- **The number of repetitions is not static.** In general, use a variable-length entity when the average number of the entity's repeating groups actually used is *less* than 75% of the maximum number of repetitions. Otherwise, use a fixed-length entity to store the repeating group. (The 75% figure is a general guideline. You should consider actual disk space savings.) See Refining the Database Design for information on fixed-length entities.

  Note: Each entity can have only one variably repeating data element.
- **SQL access to the repeating information is not a requirement.**

  If you intend to use SQL to retrieve information from the database, you may not want to create variable repeating data elements because you will not be able to access the variable portion through SQL.

You must include a counter element in the entity to indicate the current number of occurrences of the repeating data element in each entity occurrence.

If you decide to create a repeating data element in a variable-length entity, be sure to change the length of the entity on the data structure diagram. Additionally, change the storage mode of the entity to **V** (variable).



Several entities in the Commonweather database can be converted to repeating elements in variable-length entities. The NON-HOSPITAL PROCEDURE and DENTAL PROCEDURE entities should be made repeating elements because they each participate in only one relationship and occur a limited number of times:

- The NON-HOSPITAL PROCEDURE entity can be converted to a repeating element in the NON-HOSPITAL CLAIM entity.
- The DENTAL PROCEDURE entity can be converted to a repeating element in the DENTAL CLAIM entity.

Database Procedures

Database procedures are special-purpose subroutines designed to perform predefined programming functions such as data compression and decompression. You write and compile these procedures as subroutines that are executed at application runtime when a program accesses an area or entity. Database procedures have access to the entire data portion of the entity occurrence.

The time a procedure is to be called is specified in the schema. At runtime, these procedures are called automatically; the call is transparent to the application program.

Common uses

Database procedures are typically used to perform the following functions:

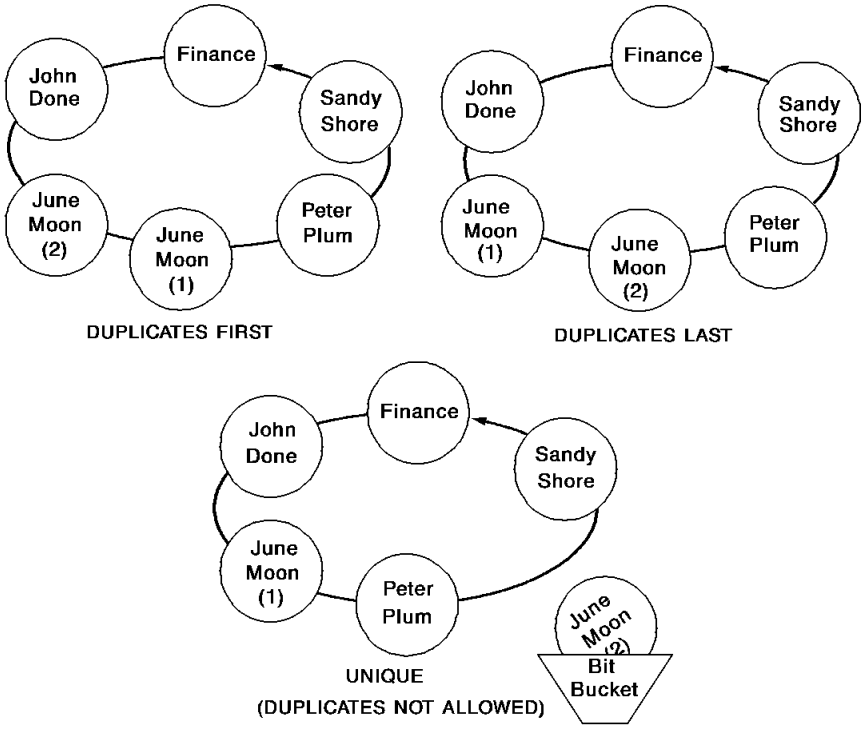
- Compression and decompression
- Data validation
- Privacy and security
- Data collection
- Determination of record length for variable-length native VSAM records

For complete information on coding and using database procedures, see *CA IDMS Database Administration Guide*.

CALC Duplicates Option

You can specify options for nonunique CALC keys indicating how these nonunique occurrences will be stored in the database. You can specify **duplicates first** or **duplicates last** .

- **Duplicates first**— The duplicate entity occurrence will be logically placed in the database *before* the entity occurrence already having that CALC key.
- **Duplicates last**— The duplicate entity occurrence will be logically placed in the database *after* the entity occurrence already having that CALC key.
- **Duplicates not allowed**— Duplicates not allowed in the non-SQL definition is equivalent to unique.



Relationship Tuning Options

There are additional tuning options available for relationships in the non-SQL environment.

Index Tuning Options

There are several index tuning options available in the non-SQL environment.

Unlinked versus Linked Indexes

An unlinked index is an index in which there are no index pointers in the child entities.

Considerations

- Unlinked indexes can be added and removed without restructuring the database, provided the control length of the entity is not changed.
- Building or rebuilding an unlinked index is faster because there are no index pointers to be maintained.
- Additional CPU and I/Os are required to locate an index entry for the current entity occurrence. For example, changing the index key value or erasing an entity occurrence both require the retrieval of the index entry.

  This additional overhead occurs because the DBMS must search the index to find the entry, whereas in a linked relationship there is a direct pointer to the SR8 occurrence containing the entry.
- Linked indexes require additional storage space.

Additional Sort Options for Indexes

Standard and natural collating sequence

You can specify either of two collating sequences for indexes:

- **Standard** collating sequence for indexes orders key fields based on their EBCDIC collating sequence without regard to data type.
- **Natural** collating sequence for indexes orders key fields based on their data type. This means that negative numeric values will collate lower than positive values.

Duplicates option

As with sorted relationships, you can order index entries with duplicate index key values as **duplicates first**, **duplicates last**, or in **db-key** sequence.

If there are many duplicates and the index is unlinked, order the duplicates by db-key. This will reduce CPU in locating a specific index entry.

Representing additional index sort options

Represent additional sort options for a relationship on the data structure diagram by specifying:

- **NATURAL** if the collating sequence is to be natural. Standard is the default.
- **DF** for duplicates first, **DL** for duplicates last, or **DBKEY** for duplicates by db-key.

Nonsorted Indexes

Nonsorted indexes are another way of linking all occurrences of an entity when the database is sparsely populated with occurrences of that entity. A nonsorted index requires less CPU and storage than a sorted index. A nonsorted index might, however, be less effective than an index sorted by db-key value. If multiple entity occurrences reside on a page, an index ordered by db-key will reduce the I/Os necessary to retrieve all occurrences.

Nonsorted orders

If the entity occurrences in the index are not to be sorted, you can specify the logical placement of new index entries by indicating one of the following orders:

- **FIRST** creates a LIFO (last in, first out) order. The new index entry is positioned at the beginning of the index.
- **LAST** creates a FIFO (first in, first out) order. The new index entry is positioned at the end of the index.
- **NEXT** creates a simple list. The new index entry is positioned immediately after the entry for the current (most recently accessed) entity occurrence. The NEXT order is recommended as a default.
- **PRIOR** creates a reverse list. The new index entry is positioned immediately before the entry for the current entity occurrence.

Index Membership Options

The same membership options are available for indexes as for relationships (see "Membership Options" earlier in this chapter).

Guidelines

Use the **mandatory-automatic (MA)** membership option unless you want only certain entity occurrences to be indexed; that is, if you want the program to control which entity occurrences are to be indexed.

Non-SQL Entity and Index Placement

To facilitate certain processing operations, you can instruct the database to store entity occurrences in a specific portion of an area (non-SQL defined databases only).

By restricting entity occurrences to a specific set of pages, you can minimize overflow conditions.

Displace a clustered entity from its owner

You can displace a clustered entity from its owner. The DISPLACEMENT clause of the non-SQL schema ADD RECORD statement allows you to store clustered entities away from their owner entity in a database area. By specifying the number of pages to displace the clustered entities, you can separate different entity types within a cluster.

Specify a subarea in which to store an entity

You can specify a subarea within an area in which a particular entity is to be stored. To separate CALC entities from other entities in an area, CA IDMS/DB allows you to assign all occurrences of a particular entity type to a range of pages.

For further information, see the WITHIN AREA clauses of the non-SQL schema ADD RECORD statement in *CA IDMS Database Administration Guide*.

Specify a subarea in which to store an index

When specifying index placement, you can specify a subarea within an area in which the owner of a system-owned index is to be stored. If you decide to place an index in an area with other database entities, you might want to assign the owner to a specific range of pages in the area.

For further information, see the WITHIN AREA clauses of the non-SQL schema ADD SET statement in *CA IDMS Database Administration Guide*.

As you plan the use of storage resources, you need to keep in mind these options for minimizing overflow conditions in the database.

Physical Tuning Options for Commonweather Corporation

Assign entities to areas

You need to assign entities to database areas to provide for efficient application runtime processing:

- **ORG-DEMO-REGION** can hold all nonclustered entities. The DEPARTMENT, OFFICE, JOB, SKILL, and INSURANCE PLAN entities can be stored in this area of the database.
- **EMP-DEMO-REGION** holds all entities clustered around the EMPLOYEE entity. The EMPLOYEE, EMPOSITION, EXPERTISE, and PROJECT entities should be stored together in this area.
- **INS-DEMO-REGION** holds all entities clustered around the COVERAGE entity. The COVERAGE, NON-HOSPITAL CLAIM, HOSPITAL CLAIM, and DENTAL CLAIM entities can be stored in this area.

By placing Commonweather entities in separate areas, we enable programs to prepare only the area or areas required for a particular operation rather than the entire database. In addition, we reduce the likelihood of contention for heavily-used entities.

You might want to assign entities and indexes to separate areas.

Compress entities

The JOB and INSURANCE PLAN entities each contain a data element that provides descriptive information about a particular entity occurrence (JOB DESCRIPTION and PLAN DESCRIPTION). As such, these entities are good candidates for compression.

Relationship options

All relationships are linked to provide most efficient access. Since this is not a large database, it is not necessary to eliminate relationships between areas.

All clustered relationships are chained; all nonclustered relationships are indexed. This reduces I/O when accessing nonclustered relationships and reduces CPU when accessing clustered relationships.

The following relationships are sorted with the unique option to eliminate indexes used only to enforce unique constraints:

New Sorted Relationship	Sort Key	Index Eliminated
EMP-EMPOSITION	START DATE	JOB-NDX
EMP-EXPERTISE	SKILL CODE	EMP-NDX

New Sorted Relationship	Sort Key		Index Eliminated
NHC-PROC	PROCEDURE	NUMBER	NON-HOSP-NDX
DC-PROC	PROCEDURE	NUMBER	PROC-NDX

The following relationships are sorted to avoid sorting retrieval occurrences:

Sorted Relationship	Sort Key
DEPT-EMPLOYEE	EMP LAST NAME EMP FIRST NAME
OFFICE-EMPLOYEE	EMP LAST NAME EMP FIRST NAME
SKILL-EXPERTISE	SKILL LEVEL

All sorted relationships are order ascending except:

- SKILL-EXPERTISE, since usually employees holding a skill should be listed such that those with the highest rating appear first
- EMP-EMPOSITION, since position START DATEs are usually increasing in value

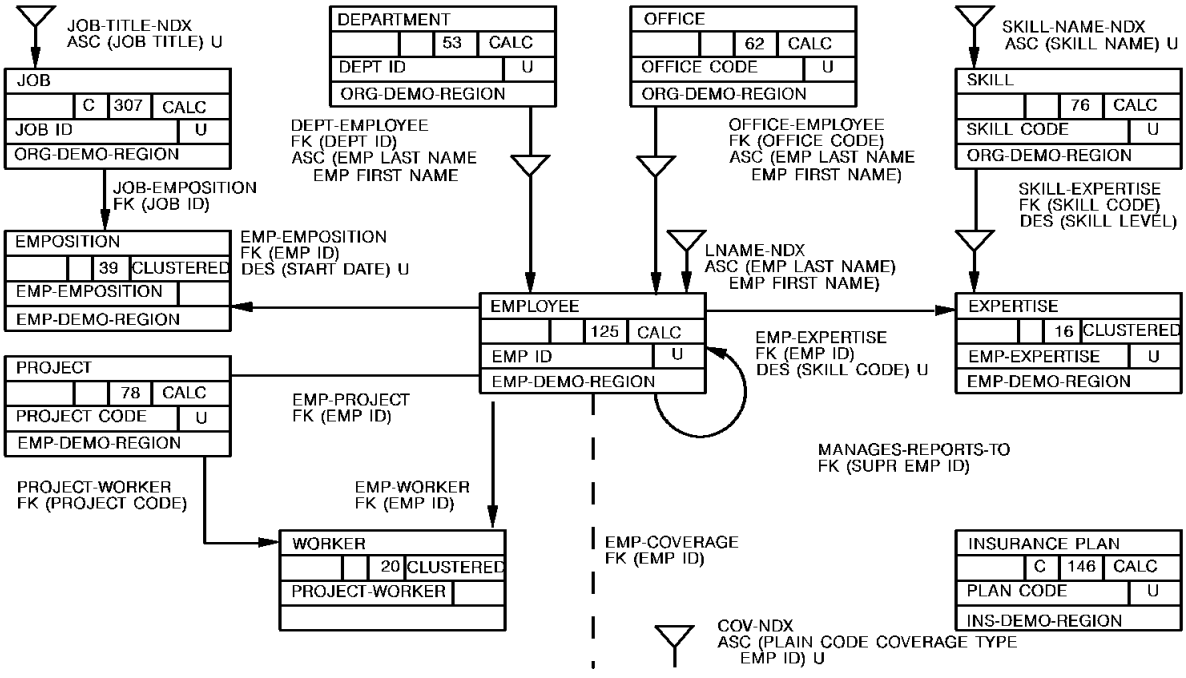
Refined Commonweather Corporation Database Design (For SQL Implementation)

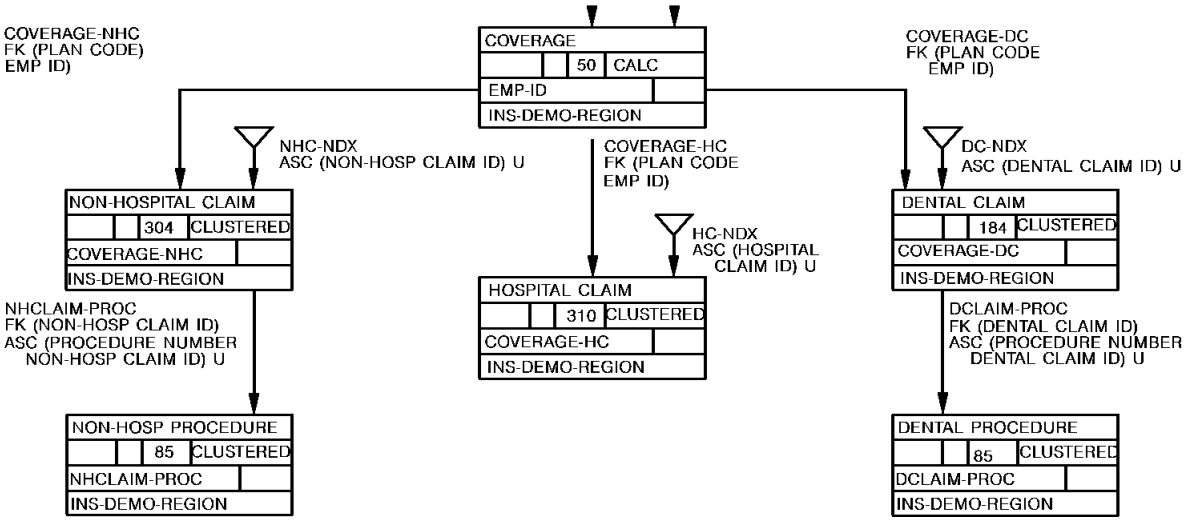
The refined data structure diagram for Commonweather Corporation (for SQL implementation) is shown below.

A review of transactions shows that all insurance information should be clustered around an employee. This can be accomplished by removing the CALC key from NON-HOSPITAL CLAIM, HOSPITAL CLAIM, and DENTAL CLAIM entities and replacing each with a unique index on NONHOSP CLAIM ID, HOSPITAL CLAIM ID, and DENTAL CLAIM ID respectively. In addition, the location mode of each of the three entities must be changed to CLUSTERED through its relationship with COVERAGE.

Due to the volume of data in the INS DEMO REGION, it is decided that all linked relationships between this region and the EMP DEMO REGION be converted to unlinked. The only relationship affected is EMP-COVERAGE. In order to convert it to unlinked, you must either add an index or CALC key on EMP ID (the foreign key of the relationship).

Since you want to cluster coverage entity occurrences by employee anyway, a CALC key on EMP ID is chosen since it achieves the same results as clustering through the EMP-COVERAGE relationship and eliminates the need for an additional index.




COVERAGE-NHC
FK (PLAN CODE)
EMP ID)
COVERAGE
50
CALC
EMP-ID
INS-DEMO-REGION
COVERAGE-DC
FK (PLAN CODE
EMP ID)
NHC-NDX
ASC (NON-HOSP CLAIM ID) U
COVERAGE-HC
FK (PLAN CODE
EMP ID)
DC-NDX
ASC (DENTAL CLAIM ID) U
NON-HOSPITAL CLAIM
304
CLUSTERED
COVERAGE-NHC
INS-DEMO-REGION
HC-NDX
ASC (HOSPITAL
CLAIM ID) U
HOSPITAL CLAIM
310
CLUSTERED
COVERAGE-HC
INS-DEMO-REGION
DENTAL CLAIM
184
CLUSTERED
COVERAGE-DC
INS-DEMO-REGION
NHCLAIM-PROC
FK (NON-HOSP CLAIM ID)
ASC (PROCEDURE NUMBER
NON-HOSP CLAIM ID) U
DCLAIM-PROC
FK (DENTAL CLAIM ID)
ASC (PROCEDURE NUMBER
DENTAL CLAIM ID) U
NON-HOSP PROCEDURE
85
CLUSTERED
NHCLAIM-PROC
INS-DEMO-REGION
DENTAL PROCEDURE
85
CLUSTERED
DCLAIM-PROC
INS-DEMO-REGION

Refined Commonweather Corporation Database Design (For Non-SQL Implementation)

Additional non-SQL physical tuning options chosen for the Commonweather Corporation database design are discussed below.

Create a multimember relationship

Since the COVERAGE, HOSPITAL CLAIM, NON-HOSPITAL CLAIM, and DENTAL CLAIM entities are usually processed together, we can create a multimember relationship to relate these entities. Let's call this relationship COVERAGE-CLAIMS.

Variable-length entities

Several entities in the Commonweather database should be converted to repeating elements in variable-length entities. The NON-HOSPITAL PROCEDURE and DENTAL PROCEDURE entities should be made variably-repeating elements because they each participate in only one relationship.

- The NON-HOSPITAL PROCEDURE entity can be converted to a repeating element in the NON-HOSPITAL CLAIM entity.
- The DENTAL PROCEDURE entity can be converted to a repeating element in the DENTAL CLAIM entity.

Add new entity

Because an employee must be managed by another existing employee, the integrity of the MANAGES-REPORT TO relationship must be ensured. In order to accomplish this in a non-SQL implementation, a new entity (STRUCTURE) and two relationships (REPORT-TO and MANAGES) must be created. Indicate the appropriate relationship options to ensure that an employee is associated with an existing employee. The MANAGES relationship is sorted to enforce unique constraints. (If it were not a sorted relationship, an index would have to be created to enforce uniqueness.)

Remove unnecessary keys

Remove foreign keys if SQL access is not a priority. If you choose to remove unnecessary keys, adjust the entity lengths accordingly.

Relationship options

Choose linkage and membership options for linked relationships. Choose ordering option of each nonsorted relationship.

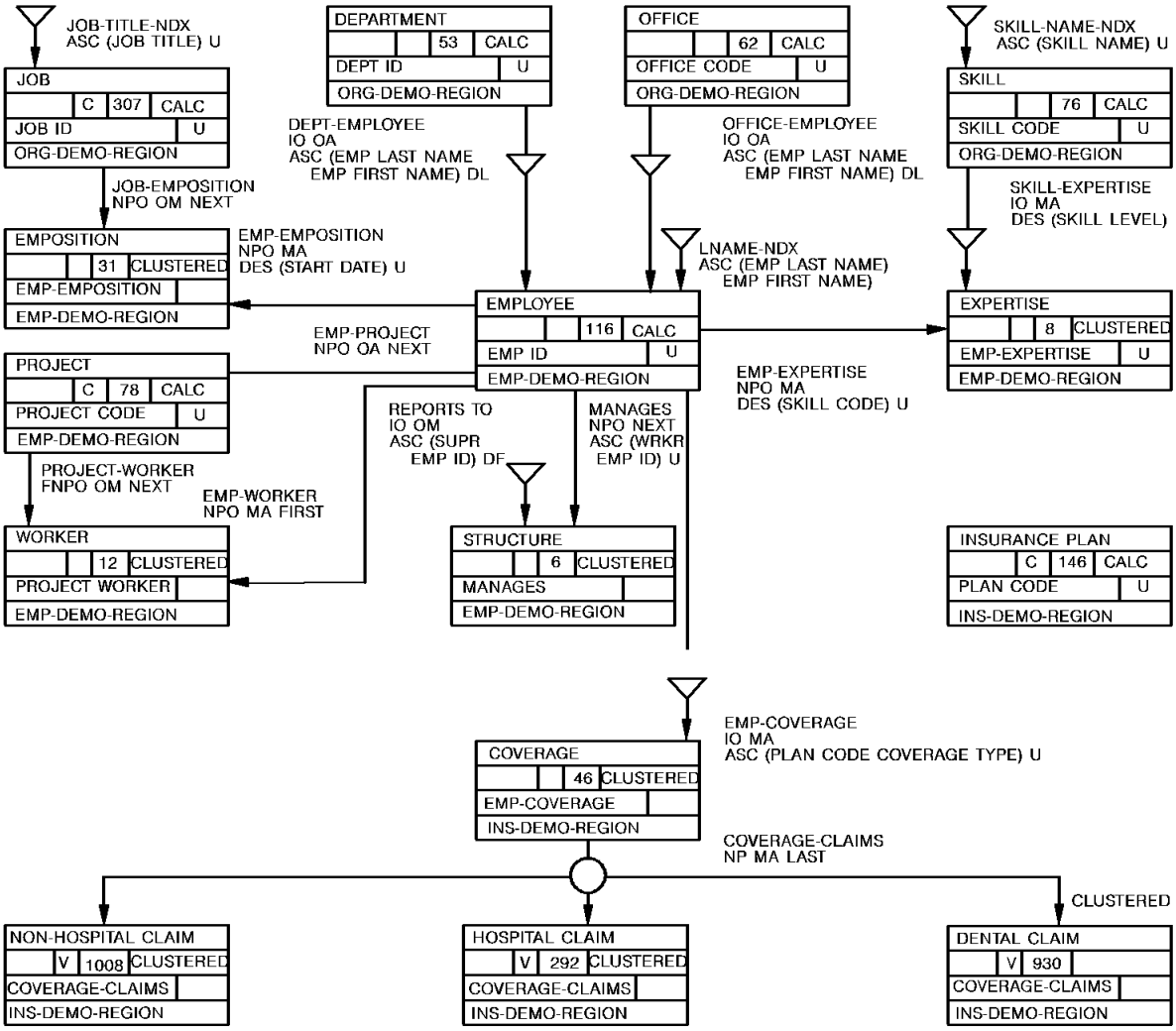
Duplicates options

Duplicates options for indexes and sorted relationships were chosen based on application requirements.

The diagram below could be used to implement the database using a non-SQL definition.

The diagram shows:

- A multimember set
- Variable-length entities
- Removal of foreign keys as reflected in new entity lengths



Since the design shown above will satisfy the performance requirements of the Commonweather Corporation, this diagram will be used in later chapters of this manual as the basis for performing sizing calculations and a final database design review.

Chapter 14: Minimizing Contention Among Transactions

This section contains the following topics:

Overview

Once you have refined the database model to optimize each individual database transaction, you should determine how the system will be affected by the concurrent execution of several transactions. You need to consider making changes to the physical model to minimize the likelihood of system bottlenecks.

Bottlenecks are often caused by excessive **contention** for database resources. For example, bottlenecks can occur when two or more programs (or terminal operators) attempt to execute update transactions against the same entity occurrences at the same time. Since the likelihood of contention increases with the number of database transactions, you need to determine whether the physical database model can accommodate the number of transactions executed at your corporation.

This chapter explains why database contention occurs and also shows you how to minimize contention.

Sources of Database Contention

Business transactions must contend for the following database resources:

- Areas
- Entities

Area Contention

Physical area locks

CA IDMS/DB examines and sets physical area locks whenever an area is opened in an update mode. Physical area locks:

- Prevent concurrent updates by multiple IDMS runtime environments (multiple local database transactions, multiple central versions, or a combination of both)
- Prevent update access to an area that requires rollback of database transactions

Physical locks are handled differently depending on the mode of processing:

- **Local mode**—As each area is readied in any update mode, the lock is checked. If the lock is set, access to the area is not allowed. If the lock is not set, the local database transaction causes the lock to be set. In the event that the transaction terminates abnormally (that is, without issuing a FINISH), the lock remains set. Further update access or commit processing by subsequent database transactions is prevented until the area is recovered.
- **Central version**—At system startup, the central version checks the locks in all areas available to the system for update processing. If any lock is set, further access to that area is disallowed (that is, the area is varied offline to the central version). The central version proceeds without the use of that area.

  If the lock is removed after system startup, the operator must vary the area status from offline to online to make the area available to the central version.

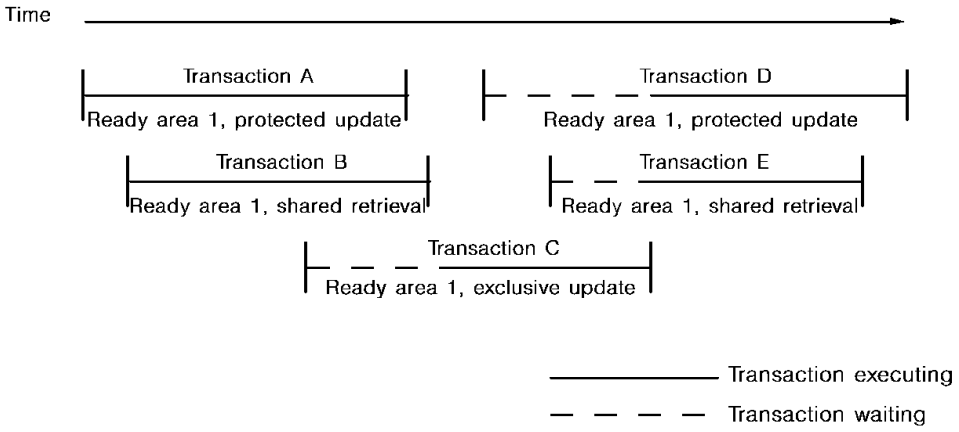
Logical area locks

Logical area locks are used by central version to control concurrent access to areas by database transactions running under central version. Logical area locks are derived from the mode in which an area is readied. A logical lock on a database area sometimes causes transactions to wait for database resources. When a transaction cannot ready an area because of a protected or exclusive restriction placed on that area by another transaction, the second transaction is placed in a wait state until the first transaction is finished.

Concurrent area access

The following diagram shows the way in which ready modes and ready options restrict concurrent use of an area by database transactions executing under one central version.

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 and is put 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 area 1, protected update
Transaction D
Ready area 1, protected update
Transaction B
Ready area 1, shared retrieval
Transaction E
Ready area 1, shared retrieval
Transaction C
Ready area 1, exclusive update
Transaction executing
Transaction waiting

Entity Occurrence Contention

Record locks

CA IDMS/DB sets record locks on entity occurrences accessed by transactions operating under the central version. Record locks are never maintained for transactions operating in local mode, since concurrent update is prevented by physical area locks.

Locks can be set implicitly by the central version or explicitly by the programmer, as described below:

- **Implicit record locks** are maintained automatically by the central version for every transaction running in shared update mode. They are optionally maintained in shared retrieval and protected update mode, according to your specifications at system generation.
- **Explicit record locks**, set by the programmer using navigational DML, are used to maintain record locks that would otherwise be released following a change in currency.

  They are never maintained for areas whose status is transient retrieval or for database transactions executing with an isolation level of transient retrieval.

Functions

Record locks perform four functions:

- Protect against concurrent update of the same entity occurrence by two or more transactions
- Protect transactions from reading uncommitted updates made by another transaction
- Protect entity occurrences that are current of one transaction from being updated by another transaction
- Allow one transaction to selectively protect any entity from access or update by another transaction

Increased contention

Record locks can sometimes increase contention among programs that require access to database resources. In some instances, conditions that result from the use of record locks can even cause abnormal termination of transactions executing under the central version. The following conditions can occur:

- **Too many locks**. If resource limits for locks are established and a transaction tries to generate more locks than the limit, the system might terminate the transaction, depending on your specifications at system generation. If resource limits for locks were *not* established, the system will continue processing, but processing performance might be degraded.

- **Excessive wait time**. If a transaction, while attempting to set a record lock, is made to wait for another transaction to terminate (or to release a lock on an entity), the first transaction waits only as long as the interval specified at system generation before abending. When a transaction exceeds the internal wait time, the system will terminate the transaction.
- **Deadlock situation**. If two transactions are in a deadlock, one of the transactions is aborted. A deadlock occurs when two transactions wait on each other for access to the same resource(s). For example, if both transaction A and transaction B read the same entity occurrence, each acquires a shared record lock on the occurrence. If transaction A then tries to update the entity occurrence, it will wait until transaction B releases its lock. If transaction B tries to update the occurrence, it will wait on transaction A. Transactions A and B are in a deadlock situation.

  CA IDMS/DB resolves this potential bottleneck by aborting and rolling back one of the transactions. By default, the transaction chosen is the most recently begun transaction with the lowest priority.

Minimizing Contention

Guidelines

You can reduce the likelihood of bottlenecks resulting from area and entity occurrence contention by making appropriate changes to the physical database design. To make intelligent design decisions to reduce contention, you must first identify potential bottlenecks.

Chapter 10, "Identifying Application Performance Requirements" showed you how to determine:

- The priority of each business transaction
- The frequency of execution of each transaction
- The frequency of access of each entity

By examining this information closely, you can identify potential bottlenecks in the physical database. For example, if you know that two different database entities will be accessed often, you can assign these entities to different areas to avoid area contention. Additionally, you can schedule the execution of high-priority programs to reduce the likelihood of contention with other programs.

Minimizing Contention for Entities and Areas

Guidelines

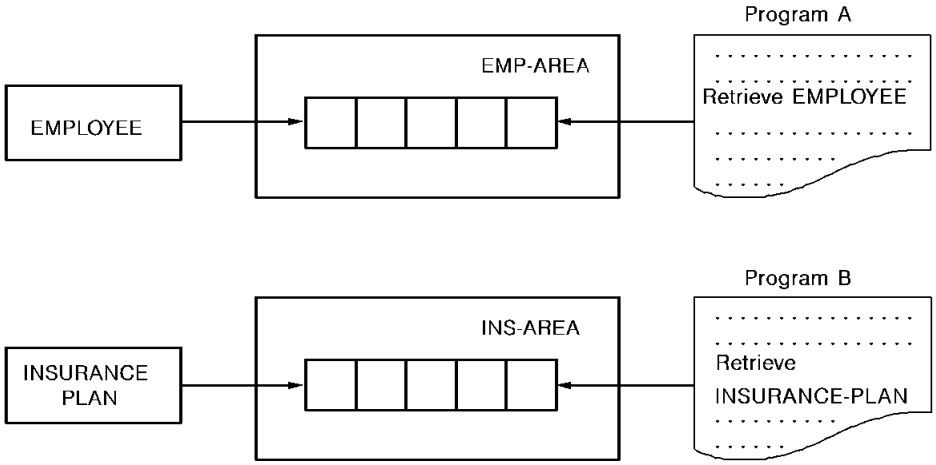
Consider the following guidelines for minimizing contention for database entities and areas:

- **Minimize the use of one-of-a-kind (OOAK) entities**.

  To reduce contention for an OOAK entity used for maintaining a control number (like the next order number in an order-entry system), you can manufacture the control number. For example, instead of storing the number in the database, you could determine the number dynamically from the date and time at which each order is placed.

- **Avoid placing heavily-used entities in the same area**. If several heavily-used entities are placed in the same database area, the area may become a source of database contention. When heavily-used entities are stored in the same area, programs may have to contend for storage space, and internally-maintained control structures such as those used for CALC processing.

  To minimize area contention, you can assign each heavily-used entity to a separate area in the database, as shown in the following diagram.

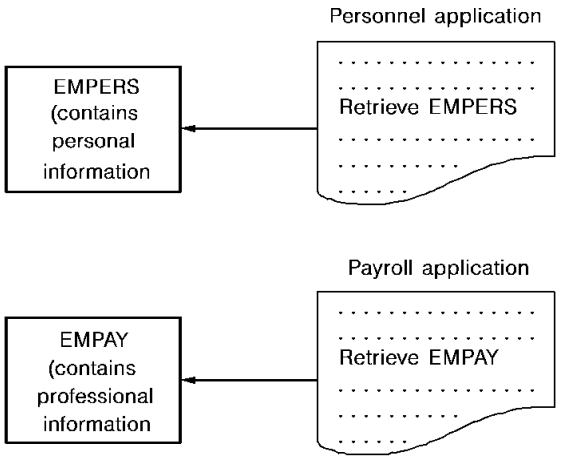


For further information on assigning entities to database areas, see Chapter 15, "Determining the Size of the Database".

- **Place large indexes in separate areas**. To avoid contention for space and because indexes are typically heavily used, place them in separate areas.
- **Avoid long-running update transactions**. Application programs that perform many updates often set many record locks. To lessen the possibility of abnormal termination as a result of setting too many locks or being involved in a deadlock, the programmer can commit database changes to release locks at intervals throughout the processing.

  This technique should be used with caution, since the commit function also causes a checkpoint to be written to the journal file. Following the unsuccessful execution of a DML function, a transaction is rolled back only to the point of the last checkpoint. Thus the existence of a checkpoint resulting from a commit statement would prohibit the system from performing a rollback to the beginning of the transaction.
- **Separate frequently used and updated entities**. If an entity creates excessive contention among application programs, you can segment the entity into two or more entities. For example, if the EMPLOYEE entity were a source of contention, you could break the entity into EMPERS and EMPAY. EMPERS might contain all personal information about each employee, while EMPAY could contain professional information. The two entities could then be assigned to different database areas and use different indexes.

  By segmenting employee data, you could eliminate contention between those programs that access employee personal information and those programs that only require access to professional information, as shown in the following diagram.



- **Include several levels for each frequently-updated sorted index**. While sorted indexes with very few levels can be used to optimize performance in retrieval applications, they sometimes cause contention between application programs that perform update functions.

  If a sorted index will be updated frequently, make sure that the index consists of at least three levels. For further information on sizing a sorted index, see Chapter 15, "Determining the Size of the Database".

- **Schedule the execution of batch update jobs**. In some situations, you should consider scheduling programs that execute batches of updates to reduce contention. By executing update programs one at a time, you can ensure that these programs do not have to contend for the same database resources.

- **Ready areas in shared update mode**. If an application program readies an area in protected or exclusive mode, other programs can be placed in a wait state. Therefore, whenever possible, programs updating a limited number of entities before a commit should ready areas in shared update mode. The shared update mode allows multiple transactions under the same central version to access the area concurrently, thereby reducing area locking and contention.

Chapter 15: Determining the Size of the Database

This section contains the following topics:

Overview

After you have decided how each entity in the database will be stored and accessed, you can determine how much storage space to reserve for the database. To allow for the most efficient processing, you need to plan the best use of available computer storage resources.

As you determine the size of the database, you need to consider several factors, including the hardware available at your corporation and the type of business applications that will be using the database.

After presenting a discussion of general database sizing considerations, this section shows you how to:

- Calculate the size of an area
- Allocate space for indexes
- Place areas in files
- Size a megabase

General Database Sizing Considerations

Before you determine the size of the database, you need to be familiar with the following topics:

- Sizing considerations for variable-length entities
- Space management for areas
- Overflow conditions
- Assignment of entities to areas
- Assignment of areas to buffers

Sizing Considerations for Compressed and Variable Length Entities

Internally, the DBMS treats the following types of entities as variable in length:

- **Fixed-length compressed entities**—Entities with a fixed length that are compressed through a compression routine; although the length of these entities is fixed from the point of view of user programs, compression makes them internally variable.
- **Variable-length entities**—Compressed or uncompressed entities with a length that depends on a variably occurring data element (that is, entities that contain an OCCURS DEPENDING clause).

Fragmentation

The DBMS fragments a variable-length entity occurrence when it is unable to store the entire entity on a single page. Fragmentation forces the system to perform two or more I/Os to retrieve a single variable-length entity. Fragmentation should be kept to a minimum.

Root and fragment size

In a non-SQL environment, you can specify the following information in the schema:

- **Minimum root**— The smallest amount of data to be stored on the entity's home page (target page)
- **Minimum fragment**— The smallest amount of data to be stored on any additional page

For SQL compressed entities, the minimum root and fragment are assigned automatically.

If a variable-length root or fragment exceeds 30 percent of the page size, space management problems can occur. To ensure efficient space management, you need to tailor the size of the minimum root and fragment to the optimal page size for the database area.

Page reserve

When a database area contains variable-length entities, and a general increase in the size of the entities is anticipated, you should define a **page reserve** in the area definition. By specifying a page reserve, you can minimize fragmentation of variable-length entities.

The page reserve is a specified number of bytes per page that can be used only for expansion of variable-length entities or internally-maintained index records. For further information on internally-maintained index records, see "Allocating Space for Indexes" later in this chapter. The space will not be used for storing new entity occurrences. In general, **page reserve should always be less than 30 percent of the page size**.

The page reserve is specified in the CREATE/ALTER AREA statement of the physical database definition.

Note: A page reserve does not affect the physical structure of the database. You can, therefore, vary the page reserve by using different DMCL modules, each with a different page reserve.

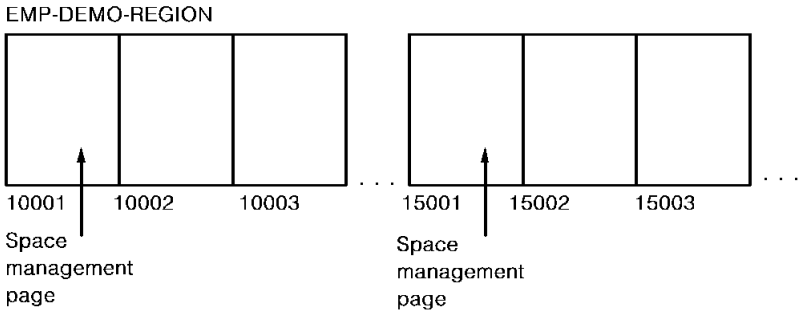
More Information For more information on the physical database definition, see the *CA IDMS Database Administration Guide*.

Space Management

To manage space in an area, the DBMS keeps track of available space on each page. CA IDMS/DB reserves selected pages called **space management pages (SMPs)** for this purpose.

Space management pages

The first page in each area is an SMP; depending on the number and size of pages in the area, CA IDMS/DB can reserve additional SMPs throughout the area. When you determine the size of an area, you need to take into consideration the number of SMPs to be maintained in the area.



More information For more information on space management, see the *CA IDMS Database Administration Guide*.

Overflow Conditions

Overflow conditions occur when entities must contend for storage space in the database. In some instances, overflow can cause performance degradation. Therefore, you need to understand the causes of overflow and know how to minimize it.

You should try to predict the effectiveness of segregating entities in the planning stage and then fine tune the database in a test environment.

Note: You can use the database analysis utility (IDMSDBAN) to determine the total number of overflows in a database.

Types of overflow

There are two types of overflow:

- CALC overflow
- Cluster overflow

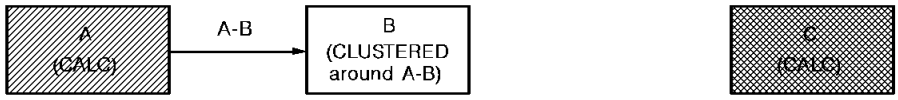
Each of these types of overflow is discussed separately below.

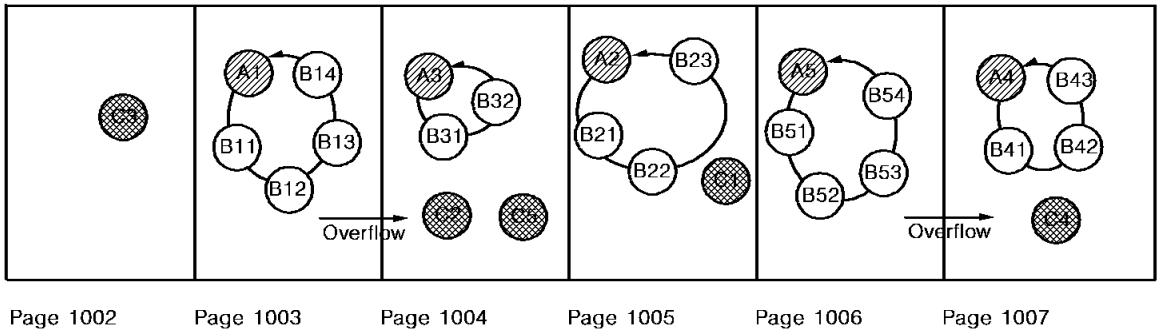
CALC Overflow

If occurrences of several entity types are randomized in one area or if an insufficient number of pages exists for the number of occurrences of one CALC entity type, CALC overflow conditions can occur.

Suppose an area contains two CALC entity types, A and C, and one clustered entity type, B, that is clustered through the A-B relationship. One A and four B entities fill a page, so that in several instances there is no room for a C entity randomizing to the same page. CALC overflow can occur in this situation, as shown below.

In this instance, A and B entities have filled pages 1003 and 1006, and have caused C2 and C4 to overflow to the next page. Two accesses are required to retrieve these entities.





Some overflow should be expected. Be concerned if a high percentage (more than 25%) of CALC entities overflow.

Reducing overflow

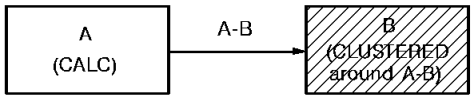
To reduce overflow:

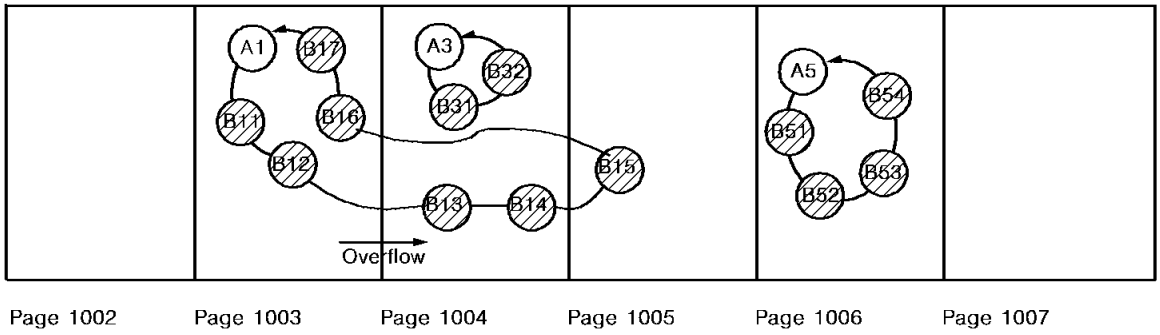
- Ensure non-static areas are no more than 75% full.
- Initially load CALC entity occurrences before clustered entity occurrences. (This is especially effective in static databases.)
- Separate entities into different areas.

Cluster Overflow

If the page size for a database area is not large enough to hold an entire cluster of entity occurrences, **cluster overflow** conditions may occur. Cluster overflow occurs when the DBMS cannot fit a new entity occurrence on the same page as other entity occurrences in the cluster. Cluster overflow forces the DBMS to try to store the entity occurrence on the next page in the area.

Suppose an area contains one entity, A, stored CALC, and one entity, B, which is clustered through the A-B relationship. One A and four B occurrences fill a page. In the instance shown in the diagram, one of the A-B clusters contains two B occurrences, one contains four occurrences, and one contains seven occurrences. Since there isn't room for the seven occurrences on one page, the extra occurrences have had to overflow to pages 1004 and 1005. To retrieve all occurrences in the cluster requires three accesses.





Reducing cluster overflow

You can reduce cluster overflow by:

- Increasing the page size for the area
- Assigning clustered entities to separate areas from their parent entities

Calculating the Size of an Area

To determine the amount of space necessary for a particular database area, you need to perform the following procedures:

1. Calculate the size of each cluster.
2. Determine the page size.
3. Calculate the number of pages in the area.

Follow steps 1 through 3 as described below to determine the size of the areas in your database.

Step 1: Calculating the Size of Each Cluster

Through clustering, users can store related entities close together in the database. Clustering allows a business application to access related entities quickly and efficiently. To ensure optimal processing, you need to base your database sizing calculations on the size of a cluster.

If you don't plan the use of storage resources effectively, the system may be unable to fit an entire cluster on a single page. Overflow conditions may occur, causing the system to perform two or more I/Os to access each application cluster. For a detailed discussion of overflow conditions, see "Overflow Conditions" earlier in this section.

Procedure

You can use the following procedures to calculate the size of a cluster:

1. Identify the entity types in the cluster.
2. Determine the length (in bytes) of each entity type stored in the cluster.
3. If an entity participates in a relationship, add 4 bytes for each NEXT, PRIOR, OWNER, or INDEX pointer.

   Note: In an SQL implementation, linked clustered relationships always contain NEXT, PRIOR, and OWNER pointers. Linked indexed relationships always contain INDEX and OWNER pointers.
4. If an entity in a non-SQL implementation is indexed, add 4 bytes for the INDEX pointer associated with each **linked** index.
5. If an entity is stored CALC, add 8 bytes to allow for pointers in the CALC (SR1) chain.
6. If an entity is variable length or compressed, add 8 bytes to allow for the variable-length indicator and fragment pointer.
7. Add 8 bytes for each entity to allow for storage of line indexes.
8. Sum the numbers calculated above to determine the total number of bytes for a single occurrence of each entity type.
9. Determine the average number of occurrences of each entity type in a single cluster.
10. Multiply the total bytes for each entity by the number of occurrences in the cluster to calculate the amount of space needed for each entity type in the cluster.
11. Add the above space calculations to determine the total size for a single cluster.

Note: If any entity in the cluster is the parent of an indexed relationship, you need to allow space for storage of the internal index entities.

Sample cluster size calculation

The following diagram shows how the size of a cluster is determined.

In the EMP-DEMO-REGION area, 508 bytes will be required to store a complete cluster of EMPLOYEE, EXPERTISE, EMPOSITION, and STRUCTURE entities.

Record type	Max data length	NPOI pointers (4 bytes each)	CALC? (8 bytes)	VL or compressed? (8 bytes)	Line index (8 bytes always)	Total bytes	Nbr of occurrences	Total bytes
EMPLOYEE	128	44	8	0	8	188	1	188
EXPERTISE	16	12	8	0	8	44	3	132
EMPOSITION	40	12	8	0	8	68	2	136
STRUCTURE	20	24	0	0	8	52	1	52

Record bytes per cluster = 508 bytes

Note: If one or more indexes are to be included in the cluster, refer to the index size calculations later in this chapter.

The above calculations are for a non-SQL implementation. If this is an SQL implementation, note that the data length and index pointer options can differ.

Step 2: Determining the Page Size

Page size

Whenever possible, you should select a page size that will hold two to three clusters of data used by an application program. The maximum page size is 32764.

The following considerations apply to selecting a page size for a database area.

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 six 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
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.

Considerations

Entity size

The size of a fixed-length entity or of a variable-length entity's minimum root or fragment cannot exceed 30 percent of page size without causing additional overhead for space management. Page size should always be at least three and one-third times greater than the largest entity in the area. A higher ratio (up to ten times greater) is preferable.

Note: With a variable-length entity, the length of the root and fragment must conform to the consideration stated above. The entity itself (root plus all fragments) can be larger than the page.

Page reserve

When you calculate the page size, you need to take into consideration the amount of space necessary for the page reserve. A page reserve is used to allow space for:

- **Future growth**—At load time, you may want to reserve space in the database for storage of new data entities or for splitting of SR8 entities in an index structure. In either case, you should specify the page reserve when the database is first defined and then remove this page reserve after the database has been loaded.
- **Expansion of variable-length entities**—The page reserve for an area that contains variable-length entities is specified when the database is defined and is never removed.

Calculating the page reserve

To calculate the size of a page reserve, perform the following procedures:

1. For each variable-length entity in the area, find the difference in bytes between the anticipated starting and expanded sizes.
2. Multiply the difference for each entity type by the anticipated number of occurrences of the entity.
3. Divide the total by the number of pages in the area.

The page reserve should never exceed 30 percent of the page size.

Buffer pool size

The size of a buffer pool depends on the amount of concurrent processing to be performed against the database. To avoid excessive database I/O operations, the buffer pool should be able to hold at least five pages.

If sufficient main storage cannot be allocated for a 5-page (or larger) buffer pool, you should reduce the page size.

Suppose an installation uses type 3380 disk devices. In this environment, the main storage required to create a buffer pool of six buffer pages is:

Page Size	Main Storage Required for Six Buffers
32,764 bytes	196,584 bytes
23,476 bytes	140,856 bytes
15,476 bytes	92,856 bytes
11,476 bytes	68,856 bytes
9,076 bytes	54,456 bytes
7,476 bytes	44,856 bytes

Note: There is additional overhead for each page in the buffer pool not included in the above numbers.

Processing requirements

The number of clusters (or portions of clusters) to be stored on a page should be determined by application processing requirements:

- **For typical random processing** where direct access to data is essential, you should use small page sizes (few clusters). A small page requires less time per access and permits more concurrent processing on a channel. However, a small page also reduces the data transfer rate, causes more I/Os, and uses more disk space for a given quantity of data.
- **For typical serial processing**, large page sizes (several clusters) allow a high data transfer rate and reduce the number of I/Os. However, large pages also monopolize the channel for longer periods of time.

Page header and footer

You need to allow 32 bytes on each page for the header and footer.

Large clusters

If the size of a cluster is excessively large (greater than 1/3 to 1/2 of a track), define a new database area and move a portion of the cluster to this area. Move one or more child entities in the cluster to the new area. You can adjust the size of this new area to accommodate a large cluster by increasing the page size or by adding more pages.

Storing clusters in a separate area

When you store child entities in a cluster in a separate area from their parent entities, the position of the child entity occurrences is proportional to the position of the parent entities in their area. Therefore the sizing considerations for both areas should be similar.

Step 3: Calculating the Number of Pages in the Area

After you have identified the optimal page size for a database area, you can determine the number of pages that should be allocated to that area. If significant growth is expected early, plan for 50 percent initial capacity and allow for growth up to 70 to 80 percent. As a general rule, you should try to avoid exceeding 70 percent capacity.

Procedure

To calculate the total number of pages required for a database area, perform the following procedures:

1. Calculate the number of bytes in each entity in the area: multiply the number of bytes in each entity by the number of occurrences. Below is a form you can use to compute the number of bytes required for each entity type. After you have determined how much space is needed for each entity type, add the bytes for each entity to determine the total number of bytes for the area.
2. Calculate the number of base pages by dividing the total entity bytes by the page size minus 32.
3. Divide the result by the desired space utilization (70 percent) to get the total number of base pages. (Static files average 70 percent; dynamic files average 50 percent.) If there are any SR8 entities in the area, you may want to increase the page reserve.
4. Subtract 32 from the page size and divide by 2 (bytes per SMP entry). Divide the quotient into the number of base pages and round up to the next integer. The result is the number of space management pages.

   Note: For large databases, the CALC algorithm operates most effectively when the number of pages in the area is a prime number.
5. Add the number of base pages and space management pages to determine the total number of pages in the area.
6. To calculate the number of tracks needed, divide the number of pages in the area by the number of pages per track on the type of disk device being used.

Sample area size calculation

The following form shows how the number of pages in an area is determined.

The EMP-DEMO-REGION area needs 508,000 bytes to store all occurrences of the EMPLOYEE, EXPERTISE, EMPOSITION, and STRUCTURE entities. Calculations determine that 173 database pages of 4276 bytes need to be allocated to accommodate these entities.

Record type	Max data length	NPOI pointers (4 bytes each)	CALC? (8 bytes)	VL or compressed? (8 bytes)	Line index (8 bytes always)	Total bytes	Nbr of occurrences	Total bytes
EMPLOYEE	128	44	8	0	8	188	1	188
EXPERTISE	16	12	8	0	8	44	3	132
EMPOSITION	40	12	8	0	8	68	2	136
STRUCTURE	20	24	0	0	8	52	1	52

1. Entity bytes per area = 508 k bytes.
2. Calculate the number of base pages by dividing the total bytes by page size minus 32:

   ```
   4276 - 32 = 4244
   508,000 / 4244 = 120 pages (rounded up)
   ```

3. Divide by desired space utilization (70%): 172 (rounded up).
4. Subtract 32 from page size and divide by two. Divide the quotient into the number of base pages and round up to the next integer. The result is the number of space management pages:

   ```
   172 / 2122 = .08
   ```

   When you round up to the next whole page, only one SMP will be needed.
5. Add the number of base pages and space management pages to determine the number of pages in the area: 173
6. Divide the number of pages in the area by the number of pages per track on the type of disk device being used. The result is the number of tracks needed:

   ```
   173 / 10 = 17.3 tracks
   ```

Allocating Space for Indexes

When a database area contains an index, you must provide space in the area for storage of the index. To determine the amount of space needed, you perform some simple calculations. Before you allocate space for an index, you need to consider both the volume of data entities to be indexed and the type of internal structures that CA IDMS/DB will generate to allow access to these entities.

Following a discussion of the structure of an index, procedures for calculating the size of an index are presented below.

Index Structure

Indexes are built and maintained by the DBMS for:

- **System indexes**—These are standalone index structures providing alternate access to entity occurrences. They are defined using the OWNER IS SYSTEM clause of the non-SQL ADD SET statement or the CREATE INDEX statement in SQL.

  The root (or top entity) of a system index is an SR7 entity. This is an internal record type with a location mode of CALC. For non-SQL-defined indexes, the CALC key is the name of the index. For SQL-defined indexes, it is an internally generated name.

- **Indexed relationships**—These are index structures associated with each occurrence of a parent entity in an indexed relationship and are used to point to the associated child entity occurrences.

  They are defined using the MODE IS INDEX clause of the non-SQL ADD SET statement where the set is not defined as SYSTEM-OWNED or the LINKED INDEX clause of an SQL CREATE CONSTRAINT statement. The root of an indexed relationship is an occurrence of the parent entity.

Structure of an index

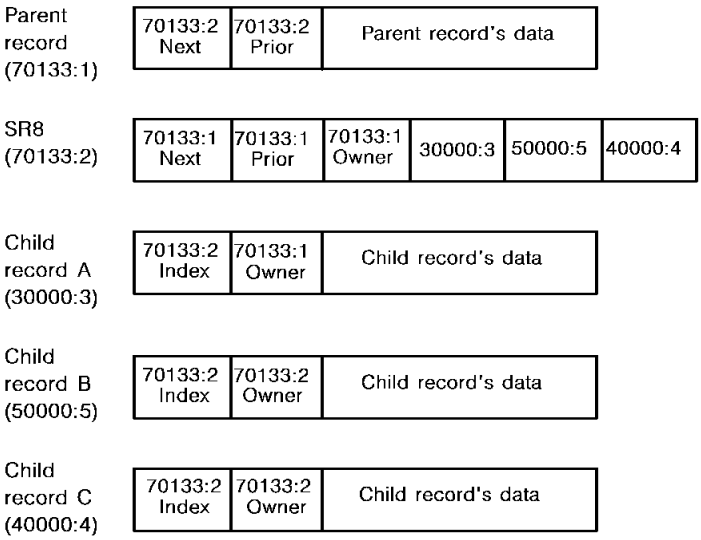
The structure of an index consists of internally maintained records called SR8s. Each SR8 is chained (by next, prior, and owner pointers) to the parent entity occurrence (or SR7 in the case of a system index) and to each other. An index is therefore structured as a chained relationship between the parent entity (or SR7) and the SR8s.

An SR8 contains from 3 to 8,180 **index entries** and a **cushion** (that is, a field that is the length of the largest possible index entry). The content of an index entry depends on the index characteristics:

- For **sorted indexes**, SR8s are arranged in levels to facilitate searching. Each index entry contains the db-key of an indexed entity occurrence *or* the db-key of another SR8. Additionally, for indexes sorted on a symbolic key, each index entry also contains a symbolic key. A symbolic key is a key constructed of one or more data elements in the order specified in the schema (up to 256 bytes in length).
- For **unsorted indexes**, SR8s are arranged in a single level. Each index entry is the db-key for an entity occurrence.

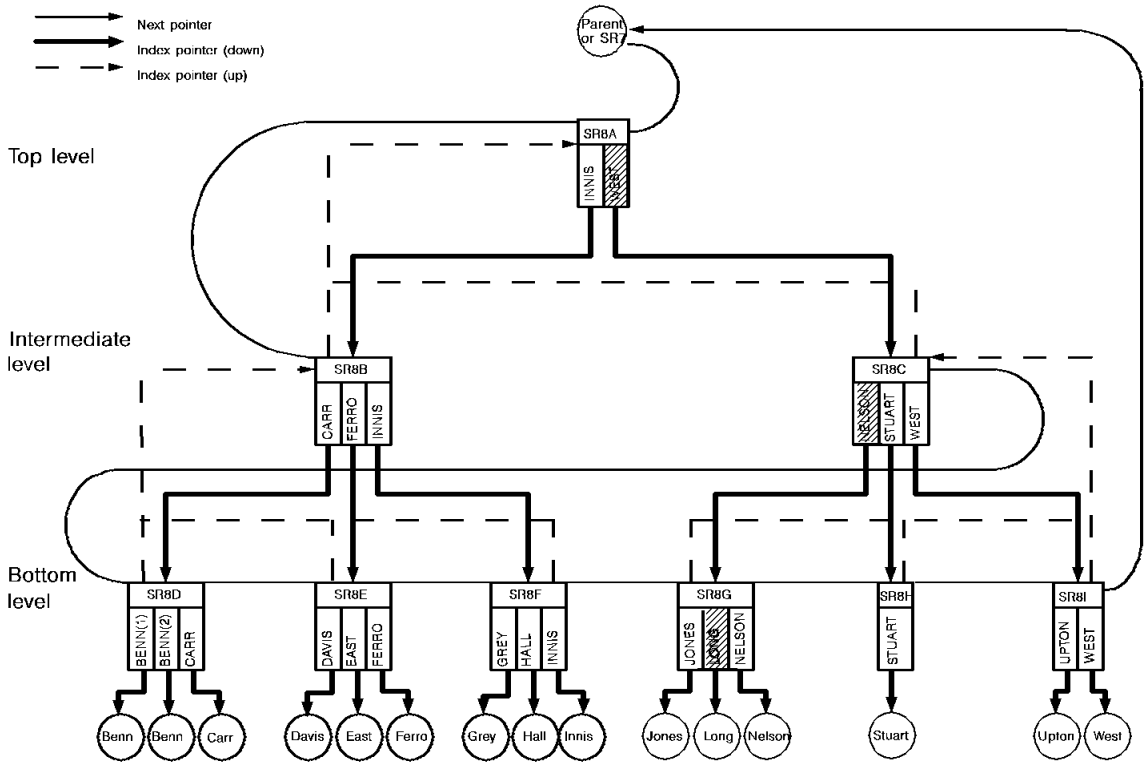
An unsorted index

The following diagram shows the structure for a simple unsorted indexed relationship. In this example, there is a single SR8 chained to the indexed set's parent. The SR8 contains three entries. Each entry contains an index pointer that points to a child entity occurrence. Each child occurrence contains an index pointer that points to that SR8 and an owner pointer that points back to the set's parent. (The owner pointer is optional.)



Structure of a three-level index

The following diagram shows the structure for a sorted index arranged in three levels. In this example, each SR8 has a maximum of three entries. Each entry consists of a symbolic key value and a db-key. The bold entries show how the LONG entity is located during an index search. In the top and intermediate levels, the db-key in each entry points to another SR8. (For simplicity, prior and owner pointers are not included in this figure.)



Entries in a 3-level index

The following diagram shows the index pointers and symbolic keys for a three-level sorted index. Each entry consists of a symbolic key and a pointer (db-key). The bold entries show how the LONG entity is located in the database. The pointers in the top and intermediate levels point to SR8s at the next lowest level. Only the bottom-level entry points to the indexed entity. (For simplicity, prior and owner pointers are not included in this figure; in addition, there are two pointers for the symbolic key for BENN, since there are two employees with that name.)

		Symbolic key	Db-key
Top-level SR8s	90002:3	Innis	90004:10
		West	**90004:57**

Intermediate-level SR8s	90004:10	Carr	90015:13
		Ferro	90016:40
		Innis	90030:6
	90004:57	**Nelson**	**90021:3**
		Stuart	90018:53
		West	90030:12
Bottom-level SR8s	90015:13	Benn	721009:147 723006:105
		Carr	721007:3
	90016:40	Davis	720617:201
		East	721592:63
		Ferro	722310:16
	00030:6	Grey	720016:31
		Hall	727160:52
		Innis	725921:74
	90021:3	James	726412:4
		Long	**724263:12**
		Nelson	727160:90
	90018:53	Stuart	720039:37
		Upton	720715:52
	90030:12	West	725129:2

Number of levels in an index

The number of levels in an index directly affects database performance. The number of levels determines:

- **The number of I/Os required to access the indexed entities**. An index that has few levels (four or fewer) typically incurs a minimum number of I/Os to access the indexed entities.
- **How much contention will occur for access to the SR8 records**. An index that has several levels typically reduces contention among application programs that require access to SR8s.

An index is considered efficient if there is little contention for the SR8s and few I/Os are required to access the indexed entities. To develop an efficient index, you should usually plan an index that has *three* levels of SR8s. An index that has more than eight or ten levels is likely to degrade processing performance by causing the system to access many SR8s when searching for a particular indexed entity occurrence. A system index that consists of fewer than three levels may incur contention if frequently updated. Indexed relationships should usually have fewer than three levels since contention is less likely because there are multiple index structures (one for each relationship occurrence).

Since the structure of an index depends on several dynamic factors, it is often difficult to make a precise calculation of the number of levels that the DBMS will create. CA IDMS/DB therefore provides schema syntax that can be used to *influence* the number of levels that will be generated for a particular index.

The number of levels generated by CA IDMS/DB for a sorted index depends on the number of index entries in each SR8. You can specify the maximum number of entries that can be contained in an SR8 by using the INDEX BLOCK CONTAINS clause of the index definition in the schema.

You can improve the efficiency of an index by performing one of the following procedures:

- **Decrease the number of levels** in the index by increasing the number of entries in each SR8. This action can enhance efficiency by decreasing the number of SR8s that the DBMS must access when searching for a particular entry.
- **Increase the number of levels** in the index by decreasing the number of entries in each SR8. This action can enhance efficiency by reducing the likelihood of contention for SR8s.

For further information on the structure of an index, see *CA IDMS Database Administration Guide*.

Calculating the Size of the Index

To account for the different types of index structures, you use a different set of formulas to calculate the size of each of the following types of indexes:

- Indexes sorted on a symbolic key
- Indexes sorted on the database key
- Unsorted indexes

Formulas for calculating the size of indexes are outlined in the following tables.

For information about sizing an index automatically, see Area statements in "Physical Database DDL Statements" of Volume 1 of *CA IDMS Database Administration Guide*.

Considerations

Before you calculate the size of your indexes, you should be aware of the following index sizing considerations:

- **The method of loading the index determines how the index size should be calculated**. The formulas presented in the tables below should be used only to calculate space requirements for indexes that are loaded in sequential order.
- **Index sizing calculations should allow ample space for future growth**. You have several options for reserving space for expansion of an index:
  - Make a generous estimate of the number of occurrences to be indexed; use this inflated number as the basis for performing your index sizing calculations.
  - Make a generous estimate of the number of pages required for the area in which the index will be stored; the formulas presented below can be used to calculate the *minimum* number of pages required for an area in which an index will be stored.
  - Specify a page reserve at load time; after the index has been loaded, remove the page reserve and increase the number of entries in each SR8.
  - Indicate how far away from the parent or SR7 the bottom-level SR8s are to be stored. For an indexed relationship or a system-owned index, you can use the DISPLACEMENT clause of the non-SQL schema ADD SET statement or the SQL schema CREATE INDEX statement to cluster bottom-level SR8s away from their parent in a database area. By specifying the number of pages to displace the bottom-level SR8s, you can reserve space in the area for storage of intermediate SR8s.

Calculating the Size of an Index Sorted on a Symbolic Key

Calculation	Formula/Instructions
Number of indexed entity occurrences and key length	The requirements of your database will determine these values. You may want to use an inflated number to allow for future growth.
Number of index levels	In most situations, you should design indexes with three levels. However, your index may consist of from one to four index levels. Indexes with few entries or a short key can be built with only two levels; indexes with many entries or very long keys might require four levels. Indexed relationships or indexes with extremely few entries might require only one level.
Number of entries per SR8	For an *n*-level index: #SR8-entries = **n**th-root-of-#indexed-entity-occurrences For example, to build a 3-level index: #SR8-entries = cube-root-of-#indexed-entity-occurrences The results of this calculation should be rounded up to the next higher integer.
Size of SR8 entities	Determine SR8 size (including line index space) by using the following formula: SR8-size = 40 + (#SR8-entries + 1) * (key-length + 8) *Key-length* equals the sum of the lengths of all data elements in the index key.

Calculation	Formula/Instructions
Number of SR8s	Determine the number of SR8s required for your index by level: #Level-0-SR8s = (#indexed-entity-occurrences + #SR8-entries - 1) ---------- #SR8-entries #Level-1-SR8s = (#level-0-SR8s + #SR8-entries - 1) ---------- #SR8-entries #Level-2-SR8s = (#level-1-SR8s + #SR8-entries - 1) ---------- #SR8-entries One of the above calculations will be required for each level in your index; note that the quotient should be truncated, not rounded. Calculate the number of SR8s at each level until the quotient equals 1. The *total* number of SR8s required for your index is equal to the sum of all the counts computed above.
Number of bytes required	Calculate the total number of bytes of space you will need to accommodate the index: Total-#bytes-required = #SR8s * SR8-size Note: *Level-0* refers to the bottom level of the index structure.

Calculation	Formula/Instructions
Page size for the index area	Plan to store at least three SR8s on a page; use a page reserve of up to 29% of each page. The page reserve factor actually increases the size of your database page so that additional SR8s can be accommodated without generating overflow. Use the following formulas to estimate page size. Page-size = (#SR8s-per-page) * (SR8-size) Total-page-size = page-size + page-reserve + page-header-footer-length The header-footer length is 32 bytes for an area. Compare the resulting page size with the table under "Step 2: Determining the page size" and select the next larger page size that's compatible with your DASD device: ■ If the page size determined in this way is too large, the number of index levels will have to be increased until a satisfactory compromise between page size and number of index levels is reached. ■ If the page size determined is much smaller than 4K, use a 4K page size instead; this allows more than three SR8s to be stored on each page.
Number of SR8 displacement pages	For improved efficiency, sorted indexes should make use of SR8 displacement pages to displace bottom-level (level-0) SR8s from the top-level and intermediate-level SR8s. To determine the number of displacement pages needed, perform these calculations: #Non-displaced-SR8s = total-#SR8s - #level-0-SR8s #SR8-displacement-pages = (#non-displaced-SR8s + #SR8s-per-page - 1) ---------- + 1 #SR8s-per-page Note that the quotient should be truncated, not rounded.
Number of pages needed for the index	After calculating the displacement pages, determine the total number of pages needed for the index: Total-#Pages-needed = #SR8-displacement-pages + (#level-0-SR8s + #SR8s-per-page - 1) ---------- #SR8s-per-page Note that the quotient should be truncated, not rounded.

Calculating the Size of an Index Sorted on db-key

Calculation	Formula/Instructions
Number of index entity occurrences	The requirements of your database will determine this value. You may want to use an inflated number to allow space for future growth.
Number of index levels	In most situations, you should design indexes with three levels. However, your index could consist of from one to four index levels. Indexes with few entries can be built with only two levels; indexes with many entries might require four levels. Indexed relationships or indexes with extremely few entries might require only one level.
Number of entries per SR8	For an *n*-level index: #SR8-entries = **n**th-root-of-#indexed-entity-occurrences For example, to build a 3-level index: #SR8-entries = cube-root-of-#indexed-entity-occurrences The results of this calculation should be rounded up to the next higher integer.
Size of SR8s	Determine SR8 size (including line index space) by using the following formulas: Level-0-SR8-size = 40 + (#SR8-entries + 1) * 4 Non-level-0-SR8-size = 40 + (#SR8-entries + 1) * 8 Round the value up to the next higher number divisible by 4. The level-0 SR8 length is nearly half that of the non-level-0 SR8. This means that a page for an index sorted on db-key can hold nearly twice as many bottom-level SR8s as higher-level SR8s.

Calculation	Formula/Instructions
Number of SR8s	Determine the number of SR8s required for your index by level: #Level-0-SR8s = (#indexed-entity-occurrences + #SR8-entries - 1) ---------- #SR8-entries #Level-1-SR8s = (#level-0-SR8s + #SR8-entries - 1) ---------- #SR8-entries #Level-2-SR8s = (#level-1-SR8s + #SR8-entries - 1) ---------- #SR8-entries The quotient should be truncated, not rounded. Continue calculating the number of SR8s at each level until the quotient equals 1. One of the above calculations will be required for each level in your index. The *total* number of SR8s required for your index is equal to the sum of all counts computed above.
Number of bytes required	Calculate the total number of bytes of space you will need to accommodate the index: #Bytes-required-for-level-0-SR8s = #level-0-SR8s * Level-0-SR8-size #Bytes-required-for-non-level-0-SR8s = #non-level-0-SR8s * non-level-0-SR8-size Total-#bytes-required = level-0-bytes + non-level-0-bytes

Calculation	Formula/Instructions
Page size for the index area	Plan to store at least three SR8s on a page; use a page reserve of up to 29 percent of the page size. The page reserve factor actually increases the size of your database page so that additional SR8s can be accommodated without generating overflow. Use the following formulas to estimate page size: Page size = (#SR8s-per-page) * (non-level-0-SR8-size) Total-page-size = page-size + page-reserve + page-header-footer-length The header-footer length is 32 bytes for a standard area. Compare the resulting page size with the result from the previous table and select the next larger page size that's compatible with your DASD device: ■ If the page size determined in this way is too large, the number of index levels will have to be increased until a satisfactory compromise between page size and number of index levels is reached. ■ If the page size determined is much smaller than 4K, use a 4K page size instead; this allows more than three SR8s to be stored on each page.
Number of SR8 displacement pages needed	For improved efficiency, sorted indexes should make use of SR8 displacement pages to displace bottom-level (level-0) SR8s from the top-level and intermediate-level SR8s. To determine the number of displacement pages needed, perform these calculations: #Non-displaced-SR8s = total-#SR8s - #level-0-SR8s #SR8-displacement-pages = (#non-displaced-SR8s + #SR8s-per-page - 1) ---------- + 1 #SR8s-per-page Note that the quotient is truncated, not rounded.
Number of pages needed for the index	After calculating the displacement pages, determine the total number of pages needed for the index: Total-#Pages-needed = #SR8-displacement-pages + (#level-0-SR8s + #level-0-SR8s-per-page - 1) ---------- #level-0-SR8s-per-page Note that the quotient is truncated, not rounded.

Calculating the Size of an Unsorted Index

Calculation	Formula/Instructions
Number of indexed entity occurrences	The requirements of your database will determine this value. You might want to use an inflated number to allow space for future growth.
Number of index levels	Unsorted indexes consist of only one level (level-0).
Number of entries per SR8	The number of SR8s should be three or more and less than the number of entity occurrences being indexed. Work out the formulas in the following steps with a number of your choice; bear in mind that you need to derive an SR8 that is less than 30 percent of the page size for the area. Recalculate the formulas as necessary until you reach the desired result.
Size of SR8s	Determine SR8 size (including line index space) by using the following formula: SR8-size = 40 + (#SR8-entries + 1) * 4 Round the value up to the next higher number divisible by 4.
Number of SR8s	Determine the number of SR8s that will be required for your index: Total-#SR8s = (#indexed-entity-occurrences + #SR8-entries - 1) ---------- #SR8-entries Note that the quotient is truncated, not rounded.
Number of bytes required	Calculate the total number of bytes of space you will need to accommodate the index: #Bytes-required-for-SR8s = #SR8s * SR8-size

Calculation	Formula/Instructions
Page size for the index area	Plan to store at least three SR8s on a page; use a page reserve of up to 29 percent of the page size. The page reserve factor actually increases the size of your database page so that additional SR8s can be accommodated without generating overflow. Use the following formulas to estimate page size: Page-size = (#SR8s-per-page) * (SR8-size) Total-page-size = page-size + page-reserve + page-header-footer-length The header-footer length is 32 bytes for an area. Compare the resulting page size with the result from the first table and select the next larger page size that's compatible with your DASD device: ■ If the page size determined in this way is too large, the number of index levels will have to be increased until a satisfactory compromise between page size and number of index levels is reached. ■ If the page size determined is much smaller than 4K, use a 4K page size instead; this allows more than three SR8s to be stored on each page.
Number of pages needed for the index	Determine the total number of pages needed for the index: Total-#pages-needed = (#SR8s + #SR8s-per-page - 1) ---------- #SR8s-per-page Note that the quotient should be truncated, not rounded.

Sample index size calculation

The following diagram shows how space is allocated for storage of an index.

The SKILL-NAME index requires 18 database pages.

For a detailed explanation of the formula used to calculate space requirements for this index, see the previous table.

```
# OF SKILL OCCURRENCES                  1680
KEY LENGTH                               12
# OF INDEX LEVELS                         3
# OF ENTRIES PER SR8                     12
SIZE OF SR8                              300
# OF SR8s                                153
# OF BYTES REQUIRED FOR INDEX          45900
# OF SR8 DISPLACEMENT PAGES                3

TOTAL # OF PAGES IN SKILL-NAME-REGION AREA  18
```

```
# SR8 ENTRIES = Cube-root-of-#skill-occurrences = 12 (rounded up)

SR8 SIZE = 40 + (13 * 20) = 300 bytes

LVL-0  =  (1680 + 11) / 12  = 140 (truncated)

LVL-1  =  (140 + 11) / 12   =  12 (truncated)

LVL-2  =  (12 + 11) / 12    =   1 (truncated)
```

(In this 3-level index, there are 140 displaced SR8s and 13 non-displaced SR8s; the total number of SR8s is 153.)

```
# OF BYTES REQUIRED FOR INDEX =   153 * 300 = 45900

SPACE REQUIRED FOR STORING 3 SR8s = 3 * 300

PAGE-SIZE (INCLUDING PAGE RESERVE) = 900/.70  + 32  = 1318
```

(1318 bytes for page size is very small; therefore a 4K page size might be used instead. If a 4K page size is selected, the DBMS will be able to store approximately 10 SR8s on a page.)

```
# OF SR8 DISPLACEMENT PAGES = (15 + 10 - 1) / 10 + 1 = 3

TOTAL # OF PAGES IN AREA    = (140 + 10 - 1) / 10 + 3 = 18
```

Placing Areas in Files

Guidelines

You can assign all areas in a database to a single file or you can distribute areas over several files. The following table provides some guidelines for assigning areas to files.

The relationship between areas and files can be defined as one-to-one, one-to-many, many-to-one, or many-to-many. Each arrangement has its advantages and disadvantages.

Relationship	Advantages	Disadvantages
One area to one file	■ Allows ease of maintenance ■ Facilitates recovery ■ Provides maximum flexibility in assigning areas to buffers	■ If used with VSAM, this arrangement can require excessive VSAM memory requirements (GETVIS).
One area to many files	■ Minimizes head/channel contention by spreading data over multiple packs ■ Optimizes processing of large and/or highly active areas	
Many areas to one file	■ Recommended for small, stable areas that are not used often	■ Restricts buffer allocations ■ Complicates DBA maintenance
Many areas to many files		■ Severely restricts buffer allocations ■ Complicates DBA maintenance ■ Minimizes flexibility in data set placement on disk ■ Complicates recovery procedures ■ Should be avoided

Processing considerations

When assigning areas to files, you should keep in mind the following processing considerations.

Input/output seek time

Follow these guidelines for minimizing seek time:

- If you need to keep all (or several) areas online, you can reduce seek time by mapping each area into files allocated across all the disk volumes.
- Place the most frequently accessed data set (database file) near the middle cylinder on a disk volume. The access arm begins a seek operation from the position where it completed the last operation; therefore, the distance the arm must travel will, on the average, be less to reach a cylinder in the middle of the disk surface.
- Place the smallest data sets that are accessed equally often near the center of the disk volume.
- When concurrently active data sets must be accessed by the same access mechanism, place them adjacent to one another.
- If possible, place small, concurrently active data sets on the same cylinder.

For more specific guidelines, consult your hardware vendor publications for the hardware devices used at your installation.

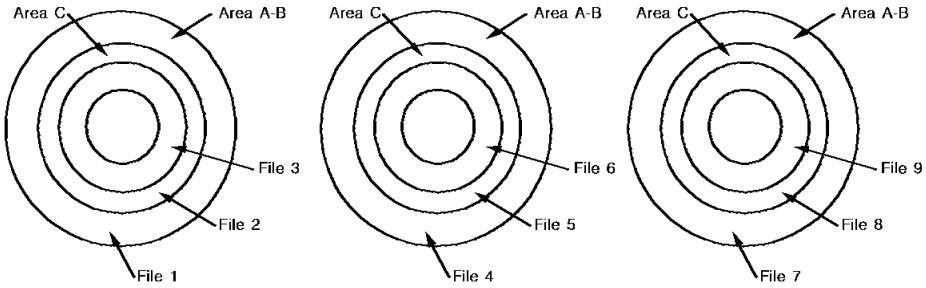
Access-arm contention

To reduce contention for use of the access arm, you can place concurrently active data sets under different access mechanisms.

Minimizing seek time

If you need to keep all areas online, you can reduce seek time by mapping the areas into files allocated across all the disk volumes. For example, you can allocate nine files, three on each volume, and map each area across all three volumes. This reduces the number of cylinders across which the disk heads must move to process any one application, as shown below.

The diagram below shows how entities used for one application can be distributed over all volumes to limit head movement.



Sizing considerations

As you assign areas to files, you need to keep in mind the following sizing considerations:

- For each page, there must be only one corresponding block of the same size.
- Pages in one area must be numbered as one continuous range of integers (you select the starting number); blocks in one file must be numbered as one continuous range of integers, starting with the number one.
- Page ranges must not overlap.
- Page size can vary from area to area but not within an area; block size can vary from file to file but not within a file. Areas with different page sizes cannot be mapped into one file, and one area cannot be mapped across files with different block sizes.
- If an area is so large that it requires more than a single physical disk device and the access method is non-VSAM, the area must be mapped to multiple files where the size of each file is no larger than the capacity of a single device.
- If VSAM is being used as the underlying access method for the database, an area of over 4GB must be mapped to multiple VSAM files.

Sizing a Megabase

To allow for processing of very large databases, CA IDMS/DB permits you to:

- Vary the format of the database key
- Assign segments to page groups

Each of these sizing options is discussed below.

Varying the Database Key Format

A database key is the concatenation of an entity's page number and its line index, for a total of four bytes. The format for a database key is variable. The page number can make up 20 to 30 bits of the database key; the line index can make up 2 to 12 bits. You determine the database key format by specifying the MAXIMUM RECORDS PER PAGE clause of the CREATE SEGMENT statement.

Since database key format is variable, you can structure the database to allow for either:

- **More pages with fewer entities per page**—The number of pages in an area can be from 2 to 1,073,741,824.
- **More entities per page with fewer pages**—Each page in a database can have from 2 to 2,727 entities.

To accommodate a very large database, you need to make sure that the highest page in an area can be expressed in the database key format. You also need to ensure that the line index is large enough to identify the highest entity occurrence on a specific page.

Note: The number specified in the MAXIMUM RECORDS PER PAGE clause indicates the *maximum* number of entity occurrences that the run-time system will place on a single page. The actual number of occurrences on a given page depends on the page size and the size of individual entity occurrences placed on the page.

Assigning Segments to Page Groups

By assigning segments to **page groups**, you can maintain, under a single central version, multiple databases that total more than a billion pages. A page group uniquely identifies a collection of page ranges. You can specify a numeric identifier in the range 0 through 32,767 as a page group.

More Information

For more information on varying the db-key and page groups, see the *CA IDMS Database Administration Guide*.

Considerations

Although segments can be assigned different page groups and database key formats, the following restrictions apply:

- By default, a single database transaction can access data in only one page group for a non-SQL-defined database. Therefore, data to be accessed together must be defined within the same page group.
- The single page group restriction for a transaction does not apply to SQL-defined databases or to non-SQL-defined databases accessed through a DBNAME with Mixed Page Group Binds Allowed. However, all records of a record type to be accessed in a single transaction must reside in the same segment. While you can horizontally segment a database, for example by placing customer information in three segments (CUSTEAST, CUSTWEST, CUSTCENT), you can access only one of these segments at a time from within a transaction.
- For non-SQL defined tables, owner and member records for a chain set must be in the same page group and have the same number of records per page.
- For SQL defined tables, referenced and referencing tables for a referential constraint must be in the same page group and have the same number of records per page, and a table and its index area must be in the same page group and have the same number of records per page.
- By default all segments accessed by a single database transaction must have the same database key format. However, when using a DBNAME with Mixed Page Group Binds Allowed, a single transaction can access data from multiple page groups, each having a different database key format.
- All segments of a dictionary must be in the same page group.

More Information

For more information on the use of Mixed Page Group Binds Allowed, see the *CA IDMS Database Administration Guide*.

Chapter 16: Implementing Your Design

This section contains the following topics:

Overview

Once you have determined the space requirements of the database, you are prepared for a final design review and implementation. You need to review the design to ensure that the database will support the business transactions performed by users at your corporation. You also need to ensure that applications that access the database will execute efficiently.

This chapter shows you how to review both the logical and physical models for a corporate database.

Reviewing the Design

Reviewing the design for a corporate database involves performing the following procedures:

1. Reviewing the logical database model
2. Reviewing the physical database model

Follow the steps below to finalize the design for your corporate database.

Step 1: Review the Logical Database Model

In the initial stages of logical design, you identified the business problem that users hoped to solve by creating a database. After interviewing several company employees, you performed a thorough analysis of the business system, determining the processing functions performed by the corporation and the flow of data during typical executions of these functions.

An analysis of the system provided documentation of the types of data required by corporate users to perform their day-to-day business tasks. With this documentation, you created the entity-relationship diagram, which serves as a model of the corporate enterprise.

During the final review of a database design, you should make sure that the physical design does not compromise the logical model for the database.

Step 2: Review the Physical Database Model

Earlier in the design process, you traced the flow of each business transaction through the database. By tracing the flow of transactions, you tried to ensure that the system would support all database processing. During the final review of a database design, you need to trace the flow of business transactions again.

Calculating I/Os

As you trace the flow of each business transaction, you should calculate the number of input/output operations that will be performed. The I/O calculation for a business transaction depends on several factors. These factors include the order in which entities are accessed, the location mode of each entity accessed, the types of indexes (if any) used to access the data, and the way entities are clustered in the database.

See Chapter 12, "Refining the Database Design" for instructions on how to estimate the number of I/Os for a transaction.

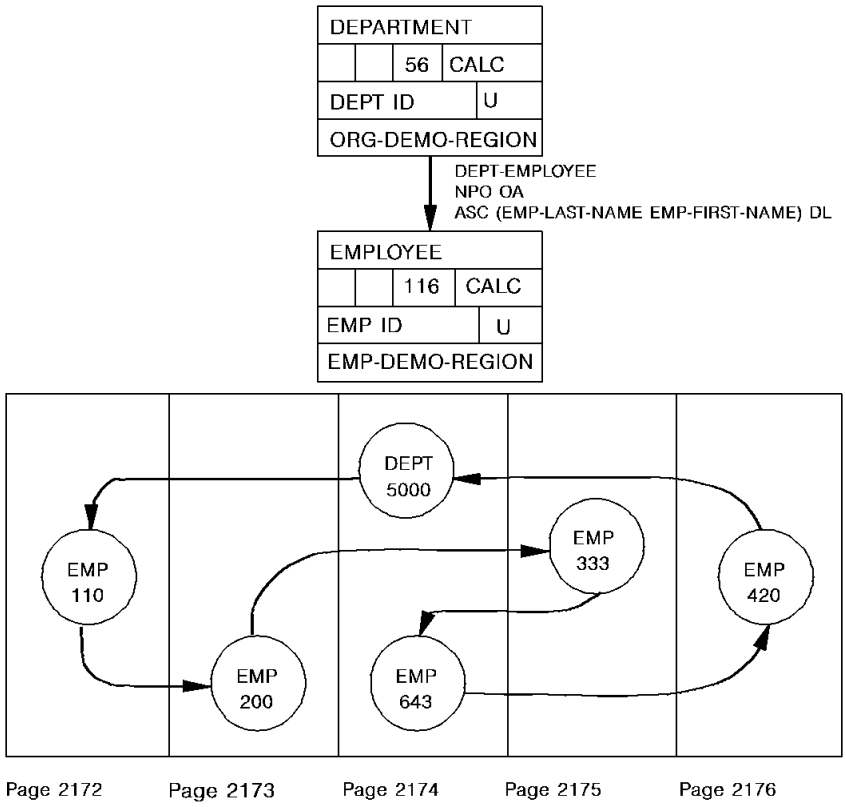
Potential Design Flaws

As you trace the flow of each transaction, you need to look for potential design flaws. Here are some things to watch out for.

Nonclustered relationships

Relationships between two entities that are stored with the CALC location mode sometimes degrade processing in applications that retrieve all child entity occurrences. When two CALC entities are related, the system must perform several I/O operations to retrieve the child entity occurrences participating in the relationship, as shown below.

CALC-to-CALC relationships are particularly costly for long chained relationships (those having many child occurrences). In the following diagram, note the number of pages accessed in order to retrieve all employees in a particular department.



Sorted relationships

Sorted relationships are efficient for some kinds of processing and not for others. When you design a relationship, you need to consider whether the sorted order is appropriate for the type of processing that will be performed.

Make sure that:

- Every sorted relationship can be justified.
- If new key values are higher than existing values, the relationship is ordered in descending sequence.
- If new key values are lower than existing values, the relationship is ordered in ascending sequence.
- If the relationship is not clustered, it is indexed rather than chained (non-SQL implementation).

For further information on sorted relationships, see Refining the Database Design.

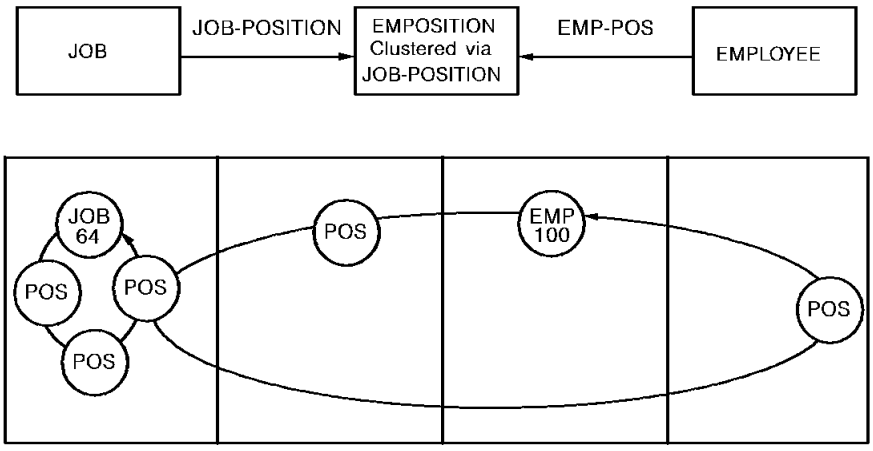
Relationships crossing areas

When two entities related through a linked relationship are stored in different database areas, certain utilities require that you operate on both areas at the same time. Therefore, you might want to consider using an unlinked relationship rather than a linked relationship.

Ineffective clustering

Processing performance can be affected by ineffective clustering. Suppose that an entity participates as a child in two relationships. To achieve optimal performance, the relationship through which an entity is most frequently accessed should be chosen as the clustering relationship.

In the example below, retrieving all positions for a job will require fewer I/Os than retrieving all positions for an employee. This should be reviewed to ensure that it reflects transaction frequencies.



Large clusters

Large clusters of entity occurrences can also cause performance problems. If the amount of space required to hold related entity occurrences is greater than the page size for a database area, CALC or cluster overflow conditions can occur.

Absence of PRIOR pointers in a non-SQL implementation

PRIOR pointers should be excluded from a relationship only when all of the following conditions are true:

- Child entity occurrences in a relationship are not erased or disconnected.
- Child entity occurrences in a relationship participate in no other relationship.
- Order is not LAST or PRIOR.
- The FIND/OBTAIN LAST or FIND/OBTAIN PRIOR functions are not used for the relationship.

In all other circumstances, you should include PRIOR pointers in a relationship.

Absence of OWNER pointers in a non-SQL implementation

OWNER pointers should be excluded from a relationship only when all of the following conditions are true:

- Parent entities in a relationship are not accessed from child entities.
- The FIND/OBTAIN OWNER DML function is not used for the relationship.
- Parent and child entities are normally stored all on one page.

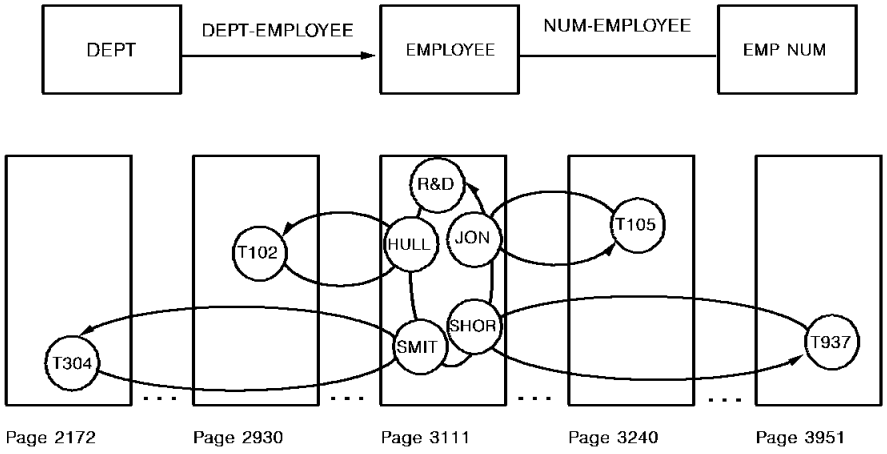
In all other circumstances, you should include OWNER pointers in a relationship. Every relationship must have NEXT pointers except indexed relationships, which must have INDEX pointers.

Questions To Address

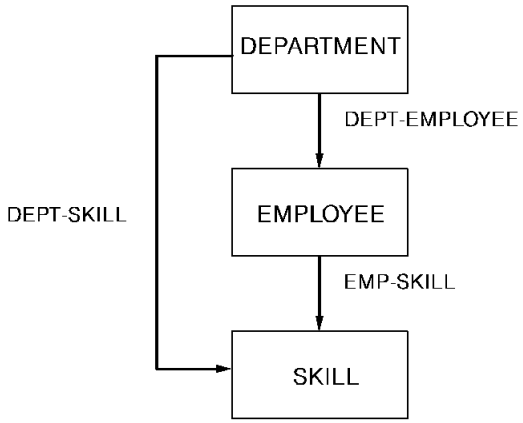
Here are some questions that you should address before implementing a database:

- **Will performance be acceptable for the five to ten most important transactions?** From a performance standpoint, the most important transactions are those transactions that are executed most frequently.
- **Do any clustered entities require rapid, random retrieval?** If so, consider placing indexes on these entities or, in a non-SQL implementation, adding additional linked relationships, as described below.

  In the following example, the EMPLOYEE entity is stored clustered via the DEPT-EMPLOYEE relationship. A new entity called EMP-NUM is created and linked to the EMPLOYEE entity in a one-to-one relationship. Using the relationship and CALC retrieval on EMP-NUM, an employee can be retrieved by employee number using two I/Os, even though it is neither a CALC nor an index key.



- **Does any entity that sparsely populates an area require processing of all occurrences?** If so, consider building an index for the entity.
- **Can extra relationships be added for more direct access?** In some cases, you might want to include additional relationships to enhance processing performance. For example, you might want to define the DEPT-SKILL relationship to allow retrieval of information from the DEPARTMENT and SKILL entities without having to retrieve employees. The diagram below shows this use of an extra relationship.

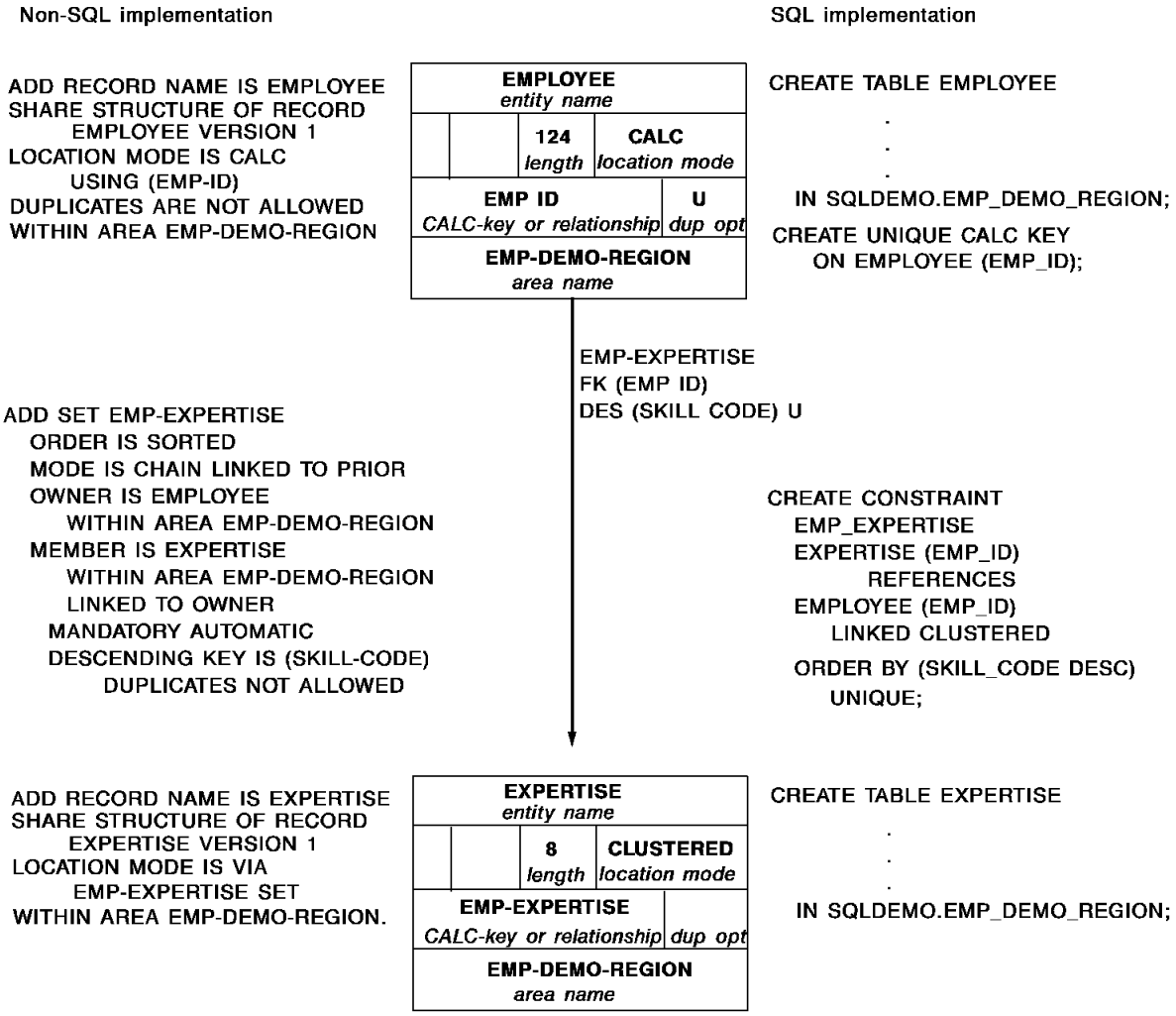


Implementing the Design

Now that you have a physical database design, it is time to implement that design. CA IDMS/DB provides two methods for implementation:

- SQL DDL statements — Available only if your site has the SQL Option
- Non-SQL DDL statements

The data structure diagram you created is used as the basis for your implementation. The diagram that follows shows a portion of the data structure, annotated with both the SQL and non-SQL definition statements that apply to the components illustrated. Complete SQL and non-SQL implementations of the Commonweather Corporation database can be found in Non-SQL Database Implementation for the Commonweather Corporation.



Implementing Your Design with SQL

You can choose to implement your design using SQL statements.

SQL terminology

The following table relates the terms used during the physical design process with those used in an SQL implementation.

Logical/Physical Design Term	SQL Implementation Term
Entity	Table
Entity occurrence	Row
Data element	Column
CALC location mode	CALC
Clustered location mode	Clustered constraint
Relationship	Constraint
Index	Index
Unique	Unique
Parent	Referenced table
Child	Referencing table

Implementation Steps

1. Decide on naming conventions for:
   - Tables
   - Columns
   - Constraints
   - Indexes
2. Create the database.
3. Create the logical definition of your database using SQL DDL statements.
4. Copy the segment definition from the system dictionary into the application dictionary in which you will define your tables.

More Information

For more information on physical definition and creation, see the *CA IDMS Database Administration Guide*.

Steps 1 through 3 are described in more detail below.

You are now ready to define the tables and other logical components of your database.

Naming conventions

Database tables and columns should have short, meaningful names. Table names are up to 18 characters in length. Columns within tables can have names of up to 32 characters. Underscores are usually used between tokens within a name (for example, SKILL_LEVEL). Hyphens should be avoided since names containing hyphens must be enclosed in double quotes when used in SQL syntax.

Referential constraints are typically named by concatenating the names of the two related tables. For example, the referential constraint between the EMPLOYEE table and the DEPARTMENT table becomes DEPT_EMPLOYEE. This convention may need to be modified, however, since constraint names can be no more than 18 characters.

Indexes must also be named. Names up to 18 characters are permitted.

Creating the database

A database is represented by a **segment**. To create a database, you:

1. Define the segment in the system dictionary using CREATE SEGMENT, FILE, and AREA statements.
2. Include the segment definition in a DMCL and punch and link edit it to a load or core image library.

3. Allocate the operating system files defined in the segment and initialize them using the FORMAT utility statement.

Creating the logical database definition

The following examples illustrate how the logical components of your design are translated into SQL DDL.

For complete DDL syntax, see *CA IDMS SQL Reference Guide*.

CREATE SCHEMA statement

A schema groups one or more tables together. Typically all tables associated with a single database, or with a specific application within a single database, are defined within one schema. The statement below defines the schema, EMPSCHM.

```
CREATE SCHEMA EMPSCHM;    ◄---------- Names the schema
```

CREATE TABLE statement

Each entity in your design is defined as an SQL table. The definition of a table includes:

- The name of the table
- A list of columns (data elements), including the data type of each, whether a default has been designated, and whether or not nulls are allowed
- An optional check constraint that limits the data that can be maintained in the database for a particular column or columns
- The name of the area in which the data for the table is to be stored

The following statement defines the table, SALARY GRADE.

```
CREATE TABLE EMPSCHM.SALARY_GRADE    ◄--------- Names the table

  (SALARY_GRADE     UNSIGNED NUMERIC(2,0)      NOT NULL, ┐
   JOB_ID           UNSIGNED NUMERIC(4,0)      NOT NULL, | Names the
   HOURLY_RATE      UNSIGNED DECIMAL(7,2)              , | columns and
   SALARY_AMOUNT    UNSIGNED DECIMAL(10,2)             , | assigns column
   BONUS_PERCENT    UNSIGNED DECIMAL(7,3)              , | characteristics
   COMM_PERCENT     UNSIGNED DECIMAL(7,3)              , |
   OVERTIME_RATE    UNSIGNED NUMERIC(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.EMP_DEMO_REGION;    ◄---- Names the area qualified
                                       with a segment name
```

Null values

SQL allows you to represent the absence of a column value in a particular row by assigning NULL to the column. This could happen because the value is not known yet (such as a credit rating when a credit check has not yet been completed for a new customer) or because it isn't applicable (such as phone number for an employee with no phone). Null values may receive special treatment in certain SQL DML statements. For example, the COUNT aggregate function doesn't include null values in a particular column when counting the number of rows based on that column.

CREATE INDEX statement

The definition of an index includes:

- The name of the index
- The name of the table and columns in the table on which the index is placed
- The area in which the index is to be stored
- The UNIQUE and/or clustering specification
- Additional physical tuning options

The statement below defines the EMP_NAME_NDX index.

```
CREATE EMPSCHM.INDEX EMP_NAME_NDX      ◄------- Names the index
    ON EMPSCHM.EMPLOYEE(EMP_LAST_NAME, EMP_FIRST_NAME) ◄-- Names the columns
    IN SQLDEMO.INDXAREA;    ◄--------- Names the area qualified with
                                       segment name
```

CREATE CONSTRAINT statement

In an SQL-defined database, relationships are the vehicle for the enforcement of referential integrity. The system automatically ensures that the foreign key columns of child rows are either null or match the primary key of an existing parent row.

Linked and unlinked relationships are implemented as constraints. The definition of a constraint includes:

- The name of the constraint
- The names of the two tables it relates
- The referenced and referencing columns
- A specification of whether the constraint is linked or unlinked
- A specification of whether child entity occurrences are to be clustered based on this relationship
- Additional tuning options

The statement below defines the EMP_EXPERTISE constraint.

```
   CREATE CONSTRAINT EMPSCHM.EMP_EXPERTISE  ◄--- Names the referential constraint

          EMPSCHM.EXPERTISE (EMP_ID) REFERENCES ┐ Names referenced and referencing
          EMPSCHM.EMPLOYEE   (EMP_ID)           ┘ tables and columns

             LINKED CLUSTERED;  ◄----- Specifies type of referential constraint
```

Creating views

SQL-defined views can be used to:

- Implement security because they can restrict access to a subset of the rows and columns within a table
- Provide a shorthand means of referring to complex SELECT statements

Below are some sample views that might be created for the Commonweather database:

```
CREATE VIEW EMPSCHM.SS_FORMAT
       (EMP_ID, EMP_LAST_NAME, EMP_FIRST_NAME, SS1, SS2, SS3)
       AS SELECT EMP_ID, EMP_LAST_NAME, EMP_FIRST_NAME,
                 SUBSTR(SS_NUMBER, 1, 3), SUBSTR(SS_NUMBER, 4, 2),
                 SUBSTR(SS_NUMBER, 6, 4)
          FROM EMPSCHM.EMPLOYEE;

CREATE VIEW EMPSCHM.EMP_HOME_INFO
       AS SELECT EMP_ID, EMP_LAST_NAME, EMP_FIRST_NAME, STREET,
                 CITY, STATE, ZIP_CODE, PHONE
          FROM EMPSCHM.EMPLOYEE;

CREATE VIEW EMPSCHM.EMP_WORK_INFO
       AS SELECT EMP_ID, START_DATE, TERMINATION_DATE
          FROM EMPSCHM.EMPLOYEE;
```

Table and view security

If CA IDMS/DB internal security is in effect, GRANT statements must be used to allow others, besides the owner, to access the tables and views within a schema. Every schema has an owner. The initial owner of a schema is the user who created it. Ownership can be transferred to another individual using the TRANSFER OWNERSHIP statement.

For more information on these statements, see *CA IDMS SQL Reference Guide*.

Implementing Your Design with Non-SQL

You can choose to implement your design using non-SQL statements.

Non-SQL terminology

The following table relates the terms used during the physical design process with those used in a non-SQL implementation.

Logical/Physical Design Term	Non-SQL Term
Entity	Record type
Entity occurrence	Record occurrence
Data element	Field/element
CALC location mode	CALC
Clustered location mode	VIA
Relationship	Set
Index	Set
Unique	Duplicates not allowed
Parent	Owner
Child	Member

Implementation Steps

1. Decide on naming conventions for:
   - Records
   - Elements
   - Sets
2. Create the logical definition of your database using non-SQL schema and subschema statements.
3. Create the database.

Each of these steps is described below.

Naming conventions

Database records and elements should have short, meaningful names. Record names are up to 16 characters in length. Elements within records can have names of up to 32 characters. Hyphens are usually used between tokens within a name (for example, SKILL-NAME).

Sets are typically named by concatenating the names of the two related records. This convention may need to be modified, however, since set names can be no more than 16 characters. For example, the set between the EMPLOYEE record and the DEPARTMENT record remains DEPT-EMPLOYEE.

Database definition

The following examples illustrate how the logical components of your design are translated into non-SQL schema statements. These statements are input to the schema compiler.

For complete DDL syntax, see *CA IDMS Database Administration Guide*.

ADD SCHEMA statement

A schema represents a logical group of records. Typically all records associated with a single database are defined within one schema.

The statement below defines the EMPSCHM schema.

```
ADD
SCHEMA NAME IS EMPSCHM VERSION 1   ◄------------ Names the schema

    SCHEMA DESCRIPTION IS 'COMMONWEATHER DATABASE'

    ASSIGN RECORD IDS FROM 1001 .
```

ADD AREA statement

Areas must be explicitly defined using the following statement.

```
ADD
AREA NAME IS EMP-DEMO-REGION   ◄-------------- Names the area

SUBAREA CALC-RANGE ◄-------------- Subarea name

SPACE  50 FROM  1  ◄-------------- Subarea page range
```

ADD RECORD statement

The definition of a record includes:

- The name of the record
- The elements included within the record (information copied from or shared with another record)
- Explicit or automatic specification of a record ID

  Record IDs are internally-used numbers assigned to each record in a schema.
- Location mode specification
- Root and fragment information for variable length records
- Optionally, database procedures to be called upon certain DML commands
- The name of the area in which this record is to be stored

The statement below defines the record EMPLOYEE.

```
ADD
RECORD NAME IS JOB   ◄-------------------------- Names the record

    SHARE STRUCTURE OF RECORD JOB VERSION 1 ◄---- Uses description of record that
 has
                                                  already been defined through ID
D
    RECORD ID IS AUTO     ◄---------------------- Instructs the system to assign
                                                  the record id
    LOCATION MODE IS CALC USING (JOB-ID)
       DUPLICATES ARE NOT ALLOWED

    MINIMUM ROOT LENGTH IS 24 CHARACTERS      ┐  Tells the system how to store
    MINIMUM FRAGMENT LENGTH IS 296 CHARACTERS ┘  fragments of this variable-lengt
h record

    CALL IDMSDCOM BEFORE STORE    ┐  Tells the system to compress the record
    CALL IDMSDCOM BEFORE MODIFY   │  during updates and decompress it for retriev
al
    CALL IDMSDCOM AFTER GET       ┘  processing

    WITHIN AREA ORG-DEMO-REGION ◄---------------- Specifies the area name
    USING CALC-RANGE            ◄---------------- and subarea
```

ADD SET statement

To implement a linked relationship, you need to define a **set**. The definition of a set includes:

- The name of the set
- The names of the owner and member records
- The linkage characteristics (index or chain) and pointer options
- Membership rules
- The set order

The statement below defines the EMP-COVERAGE set.

```
ADD
SET NAME IS EMP-COVERAGE

    ORDER IS FIRST  ◄-------------------- Tells the system to insert each new rec
ord
                                          immediately after the owner record in t
he set
    MODE IS CHAIN LINKED TO PRIOR ◄------ Tells the system that this is a chained
 set,
                                          not an indexed set and prior pointers a
re used
    OWNER IS EMPLOYEE
        NEXT DBKEY POSITION IS AUTO  ┐  Causes the schema compiler to assign poin
ter
        PRIOR DBKEY POSITION IS AUTO ┘  positions in the owner record automatical
ly

    MEMBER IS HEALCOV
        NEXT DBKEY POSITION IS AUTO      ┐  Causes the schema compiler to assign
        PRIOR DBKEY POSITION IS AUTO     |  pointer positions in the member recor
d
        LINKED TO OWNER                  |  automatically
            OWNER DBKEY POSITION IS AUTO ┘

        MANDATORY AUTOMATIC  ◄----------- Tells the system the membership option
                                          for the set
```

Subschema definition

Each subschema description for a database identifies the schema components that are available to a particular application program. Before a program containing logical record facility or navigational DML can be compiled, you must define at least one subschema.

To define a subschema, you submit the following types of statements to the subschema compiler:

- SUBSCHEMA statements
- AREA statements
- RECORD statements
- SET statements
- LOGICAL RECORD statements
- PATH-GROUP statements

A sample subschema listing for the Commonweather database is shown in Zoned and Packed Decimal Fields as IDMS Keys (see page 301).

For further information on defining subschemas, see *CA IDMS Database Administration Guide*. For further information on defining a logical record subschema, see the *CA IDMS Logical Record Facility Guide*.

Creating the database

A database is represented by a **segment**. To create a database, you:

1. Define the segment in the system dictionary using SEGMENT, FILE, and AREA statements.
2. Include the segment definition in a DMCL and punch and link edit the DMCL to a load or core image library.
3. Allocate the operating system files defined in the segment and initialize them using the FORMAT utility statement.

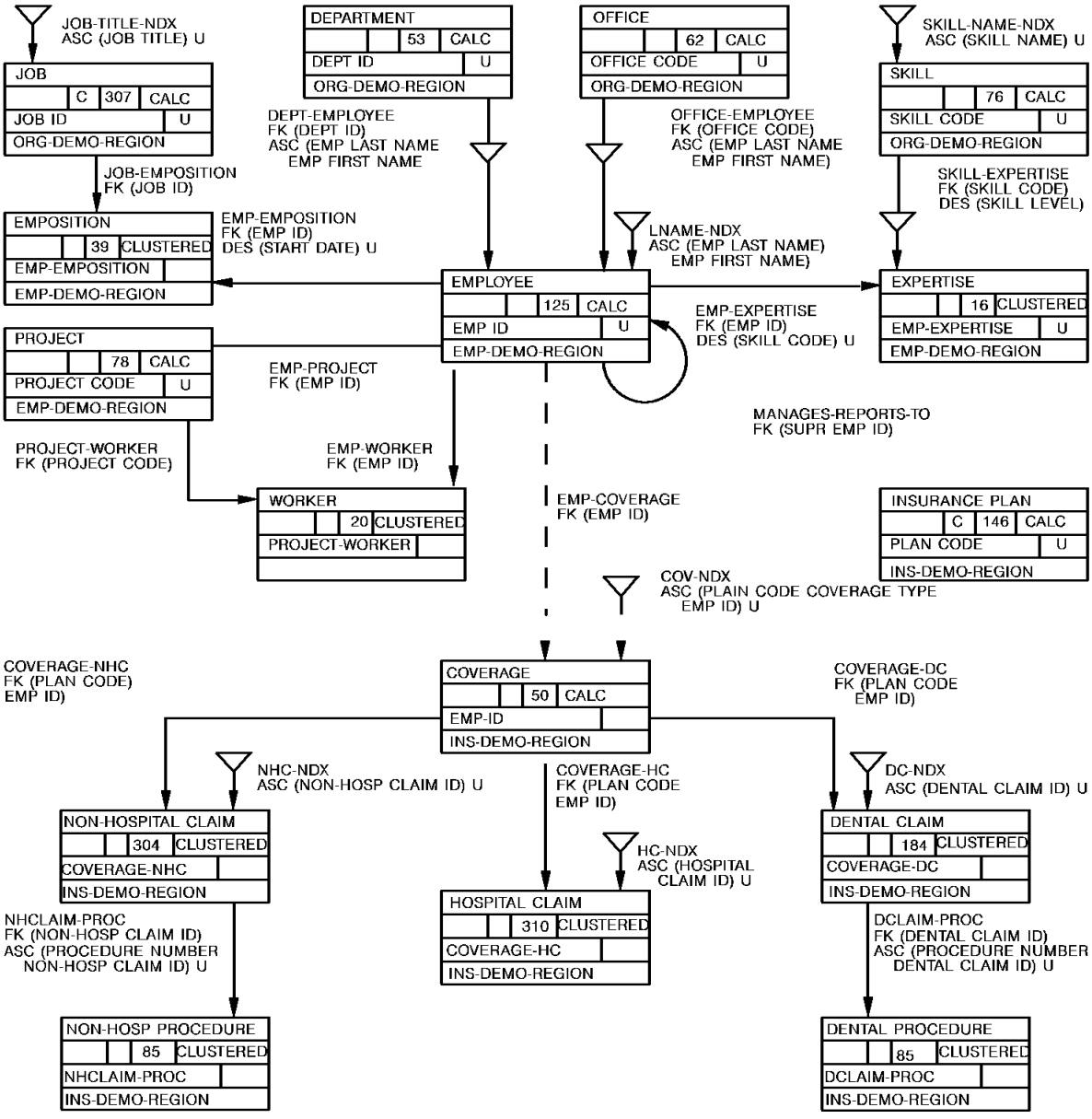
You are now ready to load data into your database.

Appendix A: SQL Database Implementation for the Commonweather Corporation

This section contains the following topics:

Logical Database Definition Listing for the Commonweather Database

Below is a listing for the SQL definition of the Commonweather Corporation database for the design shown.



Schema Statement

```
CREATE SCHEMA EMPSCHM;

SET SESSION CURRENT SCHEMA EMPSCHM;
```

Table Statements

```
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                                      )
        COVERAGE-TYPE          CHAR(01)                          NOT NULL,
     IN SQLDEMO.INS_DEMO_REGION;

CREATE TABLE DENTAL_CLAIM
       (CLAIM_DATE             DATE                              NOT NULL,
        PATIENT_FIRST_NAME     CHAR(10)                                  ,
        PATIENT_LAST_NAME      CHAR(15)                                  ,
        PATIENT_BIRTH_DATE     DATE                                      ,
        PATIENT_SEX            CHAR(01)                                  ,
        RELATION_TO_EMPLOYEE   CHAR(10)                                  ,
        EMP_ID                 UNSIGNED NUMERIC(4,0)             NOT NULL,
        PLAN_CODE              CHAR(03)                                  ,
        DENTIST_FIRST_NAME     CHAR(10)                                  ,
        DENTIST_LAST_NAME      CHAR(15)                                  ,
        DENTIST_STREET         CHAR(20)                                  ,
        DENTIST_CITY           CHAR(15)                                  ,
        DENTIST_STATE          CHAR(2)                                   ,
        DENTIST_ZIP_FIRST_FIVE CHAR(05)                                  ,
        DENTIST_ZIP_LAST_FOUR  CHAR(04)                                  ,
        DENTIST_LICENSE_NUMBER UNSIGNED NUMERIC(6,0)                     )
     IN SQLDEMO.INS_DEMO_REGION;

CREATE TABLE DENTAL_PROCEDURE
       (EMP_ID                 UNSIGNED NUMERIC(4,0)             NOT NULL,
        PLAN_CODE              CHAR(03)                          NOT NULL,
        SERVICE_DATE           DATE                              NOT NULL,
        TOOTH_NUMBER           UNSIGNED NUMERIC(2,0)                     ,
        PROCEDURE_CODE         UNSIGNED NUMERIC(4,0)             NOT NULL,
        FEE                    DECIMAL(9,2)                              ,
        DESCRIPTION            VARCHAR(60)                               )
     IN SQLDEMO.INS_DEMO_REGION;

CREATE TABLE DEPARTMENT
       (DEPT_ID                UNSIGNED NUMERIC(4,0)             NOT NULL,
        DEPT_HEAD_ID           UNSIGNED NUMERIC(4,0)                     ,
        DEPT_NAME              CHAR(40)                          NOT NULL)
   IN SQLDEMO.ORG_DEMO_REGION;
```

```
CREATE TABLE EMPLOYEE
       (EMP_ID                  UNSIGNED NUMERIC(4,0)            NOT NULL,
        EMP_FIRST_NAME          CHAR(20)                         NOT NULL,
        EMP_LAST_NAME           CHAR(20)                         NOT NULL,
        DEPT_ID                 UNSIGNED NUMERIC(4,0)            NOT NULL,
        OFFICE_CODE             UNSIGNED NUMERIC(4,0)            NOT NULL,
        STREET                  CHAR(40)                                 ,
        CITY                    CHAR(20)                         NOT NULL,
        STATE                   CHAR(02)                         NOT NULL,
        ZIP_FIRST_FIVE          CHAR(05)                         NOT NULL,
        ZIP_LAST_FOUR           CHAR(04)                         NOT NULL,
        PHONE                   CHAR(10)                                 ,
        STATUS                  CHAR(01)                         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 ('01', '02', '03', '04', '05') ) ) )
   IN SQLDEMO.EMP_DEMO_REGION;

CREATE TABLE EMPOSITION
       (EMP_ID                  UNSIGNED NUMERIC(4,0)            NOT NULL,
        JOB_ID                  UNSIGNED NUMERIC(4,0)            NOT NULL,
        START_DATE              DATE                             NOT NULL,
        FINISH_DATE             DATE                                     ,
        SALARY_GRADE            UNSIGNED NUMERIC(2,0)                    )
  IN SQLDEMO.EMP_DEMO_REGION;

CREATE TABLE HOSPITAL_CLAIM
       (CLAIM_DATE              DATE                             NOT NULL,
        PATIENT_FIRST_NAME      CHAR(10)                                 ,
        PATIENT_LAST_NAME       CHAR(15)                                 ,
        PATIENT_BIRTH_DATE      DATE                                     ,
        PATIENT_SEX             CHAR(01)                                 ,
        RELATION_TO_EMPLOYEE    CHAR(10)                                 ,
        EMP_ID                  UNSIGNED NUMERIC(4,0)            NOT NULL,
        PLAN_CODE               CHAR(03)                                 ,
        HOSPITAL_NAME           CHAR(25)                                 ,
        HOSPITAL_STREET         CHAR(20)                                 ,
        HOSPITAL_CITY           CHAR(15)                                 ,
        HOSPITAL_STATE          CHAR(2)                                  ,
        HOSPITAL_ZIP_FIRST_FIVE CHAR(05)                                 ,
        HOSPITAL_ZIP_LAST_FOUR  CHAR(04)                                 ,
        ADMIT_DATE              DATE                                     ,
        DISCHARGE_DATE          DATE                                     ,
        DIAGNOSIS               CHAR(120)                                ,
        WARD_DAYS               UNSIGNED NUMERIC(5,0)                    ,
        WARD_RATE               DECIMAL(9,2)                             ,
        WARD_TOTAL              DECIMAL(9,2)                             ,
```

```
        SEMI_DAYS              UNSIGNED NUMERIC(5,0)                  ,
        SEMI_RATE              DECIMAL(9,2)                           ,
        SEMI_TOTAL             DECIMAL(9,2)                           ,
        DELIVERY_COST          DECIMAL(9,2)                           ,
        ANESTHESIA_COST        DECIMAL(9,2)                           ,
        LAB_COST               DECIMAL(9,2)                           )
    IN SQLDEMO.INS_DEMO_REGION;
CREATE TABLE INSURANCE_PLAN
      (PLAN_CODE               CHAR(03)                       NOT NULL,
       COMP_NAME               CHAR(40)                       NOT NULL,
       STREET                  CHAR(20)                               ,
       CITY                    CHAR(15)                       NOT NULL,
       STATE                   CHAR(02)                       NOT NULL,
       ZIP_FIRST_FIVE          CHAR(05)                               ,
       ZIP_LAST_FOUR           CHAR(04)                               ,
       PHONE                   CHAR(10)                       NOT NULL,
       GROUP_NUMBER            UNSIGNED NUMERIC(6,0)          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)                  )
  IN SQLDEMO.INS_DEMO_REGION;
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(3,0)                  ,
        NUM_OPEN               UNSIGNED DECIMAL(3,0)                  ,
        EFF_DATE               DATE                                   ,
        JOB_DESC_LINE_1        VARCHAR(60)                            ,
        JOB_DESC_LINE_2        VARCHAR(60)                            ,
        REQUIREMENTS           VARCHAR(120)                           ,
        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)                  )
  IN SQLDEMO.ORG_DEMO_REGION;

CREATE TABLE EXPERTISE
       (EMP_ID                 UNSIGNED NUMERIC(4,0)          NOT NULL,
        SKILL_CODE             UNSIGNED NUMERIC(4,0)          NOT NULL,
        SKILL_LEVEL            CHAR(02)                               ,
        EXP_DATE               DATE                                   ,
   CHECK ( SKILL_LEVEL IN ('01', '02', '03', '04', '05') ) )
 IN PROJSEG.EMP_DEMO_REGION;
```

```
CREATE TABLE NON_HOSP_CLAIM
       (CLAIM_DATE              DATE                              NOT NULL,
        PATIENT_FIRST_NAME      CHAR(10)                                  ,
        PATIENT_LAST_NAME       CHAR(15)                                  ,
        PATIENT_BIRTH_DATE      DATE                                      ,
        PATIENT_SEX             CHAR(01)                                  ,
        RELATION_TO_EMPLOYEE    CHAR(10)                                  ,
       EMP_ID                  UNSIGNED NUMERIC(4,0)             NOT NULL,
        PLAN_CODE               CHAR(03)                                  ,
        PHYS_FIRST_NAME         CHAR(10)                                  ,
        PHYS_LAST_NAME          CHAR(15)                                  ,
        PHYS_STREET             CHAR(20)                                  ,
        PHYS_CITE               CHAR(15)                                  ,
        PHYS_STATE              CHAR(2)                                   ,
        PHYS_ZIP_FIRST_FIVE     CHAR(05)                                  ,
        PHYS_ZIP_LAST_FOUR      CHAR(04)                                  ,
        PHYSICIAN_ID            UNSIGNED NUMERIC(6,0)                     ,
        DIAGNOSIS               VARCHAR(120)                              )
     IN SQLDEMO.INS_DEMO_REGION;

CREATE TABLE NON_HOSP_PROCEDURE
       (EMP_ID                  UNSIGNED NUMERIC(4,0)             NOT NULL,
        PLAN_CODE               CHAR(03)                          NOT NULL,
        SERVICE_DATE            DATE                              NOT NULL,
        PROCEDURE_CODE          UNSIGNED NUMERIC(4,0)             NOT NULL,
        FEE                     DECIMAL(9,2)                              ,
        DESCRIPTION             VARCHAR(60)                               )
     IN SQLDEMO.INS_DEMO_REGION;

CREATE TABLE OFFICE
       (OFFICE_CODE             UNSIGNED NUMERIC(4,0)             NOT NULL,
        STREET                  CHAR(20)                                  ,
        CITY                    CHAR(15)                                  ,
        STATE                   CHAR(2)                                   ,
        ZIP_FIRST_FIVE          CHAR(05)                                  ,
        ZIP_LAST_FOUR           CHAR(04)                                  ,
        SPEED_DIAL              CHAR(03)                                  ,
        AREA_CODE               CHAR(03)                                  ,
        PHONE_1                 UNSIGNED NUMERIC(7,0)                     ,
        PHONE_2                 UNSIGNED NUMERIC(7,0)                     ,
        PHONE_3                 UNSIGNED NUMERIC(7,0)                     )
   IN SQLDEMO.ORG_DEMO_REGION;
CREATE TABLE PROJECT
       (PROJECT_CODE            UNSIGNED NUMERIC(4,0)             NOT NULL,
        DESCRIPTION             CHAR(40)                                  ,
        EST_BEGIN_DATE          DATE                                      ,
        ACT_BEGIN_DATE          DATE                                      ,
```

```
        EST_END_DATE            DATE                                ,
        ACT_END_DATE            DATE                                ,
        LDR_EMP_ID              UNSIGNED NUMERIC(4,0)               )
    IN SQLDEMO.EMP_DEMO_REGION;

CREATE TABLE SKILL
       (SKILL_CODE              UNSIGNED NUMERIC(4,0)          NOT NULL,
        SKILL_NAME              CHAR(20)                       NOT NULL,
        SKILL_DESC              VARCHAR(60)                           )
  IN PROJSEG.ORG_DEMO_REGION;

CREATE TABLE WORKER
       (PROJECT_CODE            UNSIGNED NUMERIC(4,0)          NOT NULL,
        EMP_ID                  UNSIGNED NUMERIC(4,0)          NOT NULL,
        BEGIN_DATE              DATE                                  ,
        END_DATE                DATE                                  )
    IN SQLDEMO.EMP_DEMO_REGION;
```

CALC Key Statements

```
CREATE UNIQUE CALC KEY ON DEPARTMENT (DEPT_ID);

CREATE UNIQUE CALC KEY ON EMPLOYEE (EMP_ID);

CREATE UNIQUE CALC KEY ON INSURANCE_PLAN (PLAN_CODE);

CREATE UNIQUE CALC KEY ON JOB (JOB_ID);

CREATE UNIQUE CALC KEY ON SKILL (SKILL_CODE);

CREATE UNIQUE CALC KEY ON PROJECT (PROJECT_CODE);

CREATE UNIQUE CALC KEY ON OFFICE (OFFICE_CODE);
```

Index Statements

```
-------------------------------------------------------------------------------
-- Create unique indexes
-------------------------------------------------------------------------------

 CREATE UNIQUE INDEX SKILL_NAME_NDX ON SKILL(SKILL_NAME);

 CREATE UNIQUE INDEX JOB_TITLE_NDX ON JOB(JOB_TITLE);

 CREATE UNIQUE INDEX COV_NDX ON COVERAGE (PLAN_CODE, COVERAGE_TYPE, EMP_ID);
-------------------------------------------------------------------------------
-- Create nonunique indexes
-------------------------------------------------------------------------------

 CREATE INDEX LNAME_NDX ON EMPLOYEE(EMP_LAST_NAME, EMP_FIRST_NAME)
        IN SQLDEMO.INDXAREA;
```

Constraint Statements

```
-------------------------------------------------------------------------
-- Create referential constraints
-------------------------------------------------------------------------

  CREATE CONSTRAINT EMP_COVERAGE
          COVERAGE  (EMP_ID) REFERENCES
          EMPLOYEE  (EMP_ID)
              UNLINKED CLUSTERED;

  CREATE CONSTRAINT DEPT_EMPLOYEE
          EMPLOYEE   (DEPT_ID)  REFERENCES
          DEPARTMENT (DEPT_ID)
              LINKED INDEX
              ORDER BY (EMP_LNAME, EMP_FNAME);

  CREATE CONSTRAINT MANAGES_REPORTS_TO
          EMPLOYEE   (SUPR_EMP_ID)  REFERENCES
          EMPLOYEE   (EMP_ID)
              LINKED INDEX;

  CREATE CONSTRAINT SKILL_EXPERTISE
          EXPERTISE  (SKILL_CODE)  REFERENCES
          SKILL      (SKILL_CODE)
              LINKED INDEX
              ORDER BY (SKILL_LEVEL DESC);

  CREATE CONSTRAINT EMP_EMPOSITION
          EMPOSITION  (EMP_ID)  REFERENCES
          EMPLOYEE    (EMP_ID)
              LINKED CLUSTERED
              ORDER BY (START_DATE DESC) UNIQUE;

  CREATE CONSTRAINT JOB_EMPOSITION
          EMPOSITION  (JOB_ID)  REFERENCES
          JOB         (JOB_ID)
              LINKED INDEX;

  CREATE CONSTRAINT OFFICE_EMPLOYEE
          EMPLOYEE  (OFFICE_CODE)  REFERENCES
          OFFICE    (OFFICE_CODE)
              LINKED INDEX
              ORDER BY (EMP_LNAME, EMP_FNAME);

  CREATE CONSTRAINT EMP_EXPERTISE
          EXPERTISE  (EMP_ID)  REFERENCES
          EMPLOYEE   (EMP_ID)
              LINKED CLUSTERED
              ORDER BY (SKILL_CODE DESC) UNIQUE;
```

```
CREATE CONSTRAINT EMP_PROJECT
        EMPLOYEE   (LDR_EMP_ID)  REFERENCES
        PROJECT    (EMP_ID)
            LINKED INDEX;

CREATE CONSTRAINT PROJECT_WORKER
        WORKER  (PROJECT_CODE) REFERENCES
        PROJECT (PROJECT_CODE)
            LINKED CLUSTERED;

CREATE CONSTRAINT EMP_WORKER
        WORKER   (EMP_ID)  REFERENCES
        EMPLOYEE (EMP_ID)
            LINKED INDEX;

CREATE CONSTRAINT COVERAGE_NHC
        NON_HOSP_CLAIM  (EMP_ID, PLAN_CODE)  REFERENCES
        COVERAGE        (EMP_ID, PLAN_CODE)
            LINKED CLUSTERED;

CREATE CONSTRAINT COVERAGE_HC
        HOSPITAL_CLAIM  (EMP_ID, PLAN_CODE)  REFERENCES
        COVERAGE        (EMP_ID, PLAN_CODE)
            LINKED CLUSTERED;

CREATE CONSTRAINT COVERAGE_DC
        DENTAL_CLAIM  (EMP_ID, PLAN_CODE)  REFERENCES
        COVERAGE      (EMP_ID, PLAN_CODE)
            LINKED CLUSTERED;

CREATE CONSTRAINT DCLAIM_PROC
        DENTAL_PROCEDURE  (DENTAL_CLAIM_ID)  REFERENCES
        DENTAL_CLAIM      (DENTAL_CLAIM_ID)
            LINKED CLUSTERED;

CREATE CONSTRAINT NHCLAIM_PROC
        NON_HOSP_PROCEDURE  (NON_HOSP_CLAIM_ID)  REFERENCES
        NON_HOSP_CLAIM      (NON_HOSP_CLAIM_ID)
            LINKED CLUSTERED;
```

Remove Default Indexes

```
ALTER TABLE COVERAGE
     DROP DEFAULT INDEX;

ALTER TABLE DEPARTMENT
     DROP DEFAULT INDEX;

ALTER TABLE EMPLOYEE
     DROP DEFAULT INDEX;

ALTER TABLE INSURANCE_PLAN
     DROP DEFAULT INDEX;

ALTER TABLE EMPOSITION
     DROP DEFAULT INDEX;

ALTER TABLE EXPERTISE
     DROP DEFAULT INDEX;

ALTER TABLE SALARY_GRADE
     DROP DEFAULT INDEX;

ALTER TABLE PROJECT
     DROP DEFAULT INDEX;

ALTER TABLE WORKER
     DROP DEFAULT INDEX;

ALTER TABLE PHONE
     DROP DEFAULT INDEX;
```

```
ALTER TABLE DENTAL_PROCEDURE
      DROP DEFAULT INDEX;

 ALTER TABLE NON_HOSP_PROCEDURE
      DROP DEFAULT INDEX;
 ALTER TABLE OFFICE
      DROP DEFAULT INDEX;

 ALTER TABLE SKILL
      DROP DEFAULT INDEX;

 ALTER TABLE DENTAL_CLAIM
      DROP DEFAULT INDEX;

 ALTER TABLE HOSPITAL_CLAIM
      DROP DEFAULT INDEX;

 ALTER TABLE NON_HOSP_CLAIM
      DROP DEFAULT INDEX;
```

View Definitions

SQL-defined views allow an application program to see just a portion of the database. A view can be used to introduce security.

Below are some sample views that might be created for the Commonweather database:

```
CREATE VIEW EMPSCHM.SS_FORMAT
       (EMP_ID, EMP_LAST_NAME, EMP_FIRST_NAME, SS1, SS2, SS3)
       AS SELECT EMP_ID, EMP_LAST_NAME, EMP_FIRST_NAME,
                 SUBSTR(SS_NUMBER, 1, 3), SUBSTR(SS_NUMBER, 4, 2),
                 SUBSTR(SS_NUMBER. 6, 4)
          FROM EMPSCHM.EMPLOYEE;

CREATE VIEW EMPSCHM.EMP_HOME_INFO
       AS SELECT EMP_ID, EMP_LAST_NAME, EMP_FIRST_NAME, STREET,
                 CITY, STATE, ZIP_CODE, PHONE
          FROM EMPSCHM.EMPLOYEE;

CREATE VIEW EMPSCHM.EMP_WORK_INFO
       AS SELECT EMP_ID, START_DATE, TERMINATION_DATE
          FROM EMPSCHM.EMPLOYEE;
```

Subschema Definition

Sample subschema listing for the Commonweather database

A sample subschema listing for the Commonweather database is shown below.

For further information on defining subschemas, see *CA IDMS Database Administration Guide*.

```
ADD
SUBSCHEMA NAME IS A200SS03 OF SCHEMA NAME IS EMPSCHM VERSION IS 1
    PUBLIC ACCESS IS ALLOWED FOR ALL
    USAGE IS MIXED
    .
 ADD
AREA NAME IS EMP-DEMO-REGION
    .
 ADD
AREA NAME IS ORG-DEMO-REGION
    PROTECTED UPDATE IS NOT ALLOWED
    EXCLUSIVE UPDATE IS NOT ALLOWED
    .
 ADD
RECORD NAME IS DEPARTMENT
    .
 ADD
RECORD NAME IS EMPLOYEE
    .
 ADD
RECORD NAME IS OFFICE
    .
 ADD
SET NAME IS DEPT-EMPLOYEE
    .
 ADD
SET NAME IS OFFICE-EMPLOYEE
    .
```

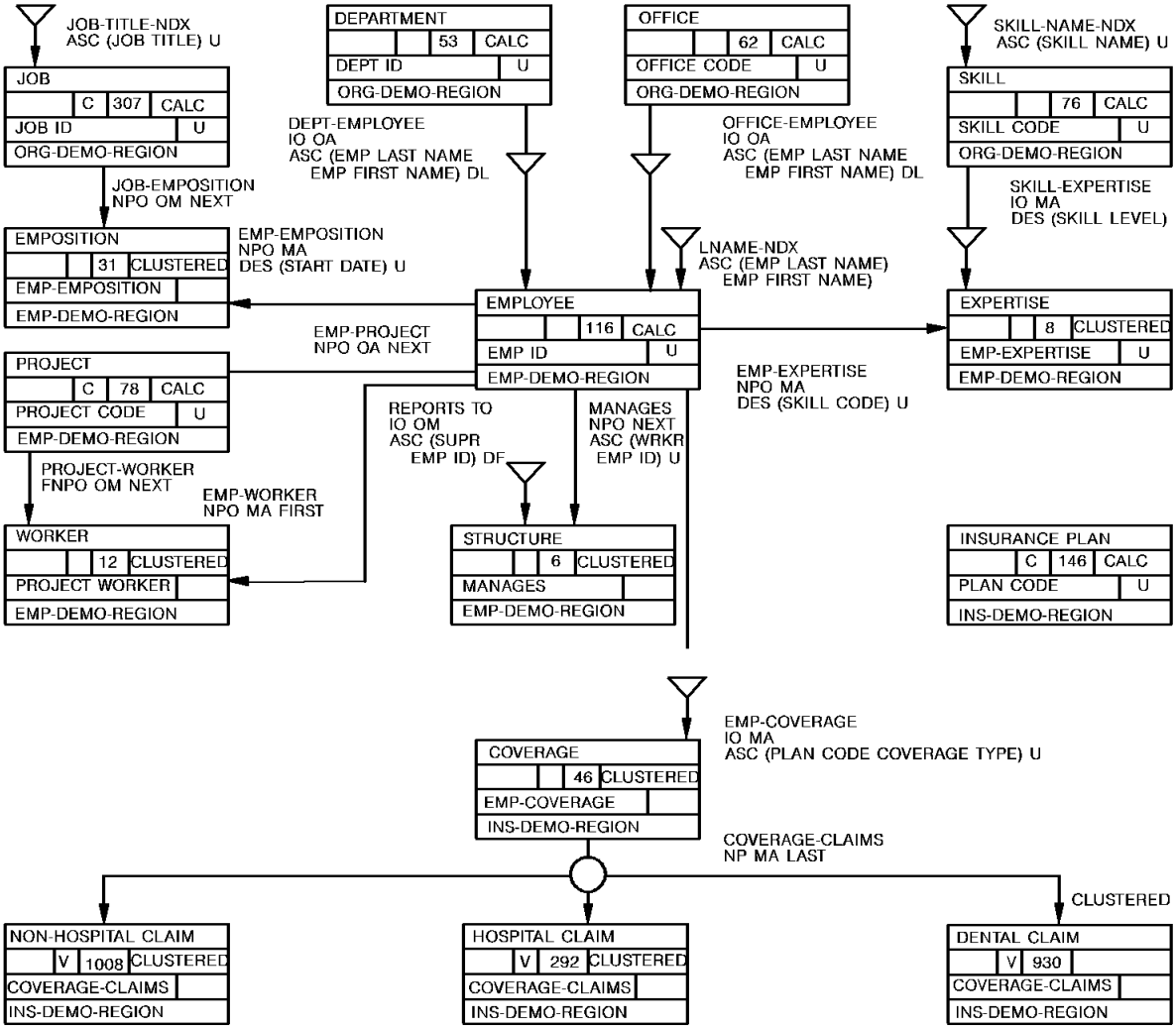
Appendix B: Non-SQL Database Implementation for the Commonweather Corporation

This section contains the following topics:

Logical Database Definition Listing for the Commonweather Database

Below is the complete non-SQL defined schema listing for the Commonweather Corporation database design shown.

Note: Once the system has assigned an ID number to each record, you should indicate this number on the data structure diagram.



Schema Statement

```
ADD
SCHEMA NAME IS EMPSCHM VERSION IS 1
    SCHEMA DESCRIPTION IS 'EMPLOYEE DEMO DATABASE'
    ASSIGN RECORD IDS FROM 1001
    PUBLIC ACCESS IS ALLOWED FOR ALL
.
```

Area Statements

```
ADD
AREA NAME IS EMP-DEMO-REGION
.
 ADD
AREA NAME IS ORG-DEMO-REGION
.
 ADD
AREA NAME IS INS-DEMO-REGION
.
```

Record Statements

```
ADD
RECORD NAME IS COVERAGE
    SHARE STRUCTURE OF RECORD COVERAGE VERSION 1
    LOCATION MODE IS VIA EMP-COVERAGE SET
    WITHIN AREA INS-DEMO-REGION.

ADD
RECORD NAME IS DENTAL-CLAIM
    SHARE STRUCTURE OF RECORD DENTAL-CLAIM VERSION 1
    LOCATION MODE IS VIA COVERAGE-CLAIMS SET
    MINIMUM ROOT LENGTH IS 132 CHARACTERS
    MINIMUM FRAGMENT LENGTH IS 930 CHARACTERS
    WITHIN AREA INS-DEMO-REGION
    .
ADD
RECORD NAME IS DEPARTMENT
    SHARE STRUCTURE OF RECORD DEPARTMENT VERSION 1
    LOCATION MODE IS CALC USING ( DEPT-ID ) DUPLICATES ARE
          NOT ALLOWED
    WITHIN AREA ORG-DEMO-REGION
    .
ADD
RECORD NAME IS EMPLOYEE
    SHARE STRUCTURE OF RECORD EMPLOYEE VERSION 1
    LOCATION MODE IS CALC USING ( EMP-ID ) DUPLICATES ARE NOT ALLOWED
    WITHIN AREA EMP-DEMO-REGION
    .
ADD
RECORD NAME IS EMPOSITION
    SHARE STRUCTURE OF RECORD EMPOSITION VERSION 1
    LOCATION MODE IS VIA EMP-EMPOSITION SET
    WITHIN AREA EMP-DEMO-REGION
    .
ADD
RECORD NAME IS EXPERTISE
    SHARE STRUCTURE OF RECORD EXPERTISE VERSION 1
    LOCATION MODE IS VIA EMP-EXPERTISE SET
    WITHIN AREA EMP-DEMO-REGION
    .
```

```
ADD
RECORD NAME IS HOSPITAL-CLAIM
    SHARE STRUCTURE OF RECORD HOSPITAL-CLAIM VERSION 1
    LOCATION MODE IS VIA COVERAGE-CLAIMS SET
    WITHIN AREA INS-DEMO-REGION
    .
 ADD
RECORD NAME IS INSURANCE-PLAN
    SHARE STRUCTURE OF RECORD INSURANCE-PLAN VERSION 1
    LOCATION MODE IS CALC USING ( PLAN-CODE ) DUPLICATES ARE
          NOT ALLOWED
    CALL IDMSCOMP BEFORE STORE
    CALL IDMSCOMP BEFORE MODIFY
    CALL IDMSDCOM AFTER GET
    WITHIN AREA INS-DEMO-REGION
    .
 ADD
RECORD NAME IS JOB
    SHARE STRUCTURE OF RECORD JOB VERSION 1
    LOCATION MODE IS CALC USING ( JOB-ID ) DUPLICATES ARE NOT ALLOWED
    MINIMUM ROOT LENGTH IS 24 CHARACTERS
    MINIMUM FRAGMENT LENGTH IS 296 CHARACTERS
    CALL IDMSCOMP BEFORE STORE
    CALL IDMSCOMP BEFORE MODIFY
    CALL IDMSDCOM AFTER GET
    WITHIN AREA ORG-DEMO-REGION
    .
 ADD
RECORD NAME IS NON-HOSP-CLAIM
    SHARE STRUCTURE OF RECORD NON-HOSP-CLAIM VERSION 1
    LOCATION MODE IS VIA COVERAGE-CLAIMS SET
    MINIMUM ROOT LENGTH IS 248 CHARACTERS
    MINIMUM FRAGMENT LENGTH IS 1008 CHARACTERS
    WITHIN AREA INS-DEMO-REGION
    .
 ADD
RECORD NAME IS OFFICE
    SHARE STRUCTURE OF RECORD OFFICE VERSION 1
    LOCATION MODE IS CALC USING ( OFFICE-CODE ) DUPLICATES ARE
```

```
            NOT ALLOWED
        WITHIN AREA ORG-DEMO-REGION
        .
    ADD
RECORD NAME IS SKILL
        SHARE STRUCTURE OF RECORD SKILL VERSION 1
        LOCATION MODE IS CALC USING ( SKILL-CODE ) DUPLICATES ARE
            NOT ALLOWED
        WITHIN AREA ORG-DEMO-REGION
        .
    ADD
RECORD NAME IS STRUCTURE
        SHARE STRUCTURE OF RECORD STRUCTURE VERSION 1
        LOCATION MODE IS VIA MANAGES SET
        WITHIN AREA EMP-DEMO-REGION
        .
    ADD
RECORD NAME IS PROJECT
        SHARE STRUCTURE OF RECORD PROJECT VERSION 1
        LOCATION MODE IS CALC USING ( PROJECT-CODE ) DUPLICATES ARE
            NOT ALLOWED
        WITHIN AREA EMP-DEMO-REGION
        .
    ADD
RECORD NAME IS WORKER
        SHARE STRUCTURE OF RECORD WORKER VERSION 1
        LOCATION MODE IS VIA PROJECT-WORKER SET
        WITHIN AREA EMP-DEMO-REGION
        .
```

Set Statements

```
ADD
SET NAME IS COVERAGE-CLAIMS
    ORDER IS LAST
    MODE IS CHAIN LINKED TO PRIOR
    OWNER IS COVERAGE
    MEMBER IS HOSPITAL-CLAIM
        MANDATORY AUTOMATIC
    MEMBER IS NON-HOSP-CLAIM
        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 INDEX BLOCK CONTAINS 30 KEYS
    OWNER IS DEPARTMENT
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
    MEMBER IS EMPLOYEE
        INDEX 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 EMP-FIRST-NAME )
            COMPRESSED
            DUPLICATES ARE LAST
    .
 ADD
SET NAME IS EMP-COVERAGE
    ORDER IS SORTED
    MODE IS INDEX BLOCK CONTAINS 30 KEYS
    OWNER IS EMPLOYEE
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
    MEMBER IS COVERAGE
        INDEX DBKEY POSITION IS AUTO
        LINKED TO OWNER
            OWNER DBKEY POSITION IS AUTO
        MANDATORY AUTOMATIC
    ASCENDING KEY IS ( PLAN-CODE COVERAGE-TYPE )
        DUPLICATES NOT ALLOWED
    .
```

```
ADD
SET NAME IS EMP-EMPOSITION
    ORDER IS SORTED
    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

    DESCENDING KEY IS ( START-DATE )
        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 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-CODE )
            DUPLICATES ARE NOT ALLOWED
    .
 ADD
SET NAME IS LNAME-NDX
    ORDER IS SORTED
    MODE IS INDEX BLOCK CONTAINS 40 KEYS
    OWNER IS SYSTEM
    MEMBER IS EMPLOYEE
        INDEX DBKEY POSITION IS AUTO
        OPTIONAL AUTOMATIC
        ASCENDING KEY IS ( EMP-LAST-NAME EMP-FIRST-NAME )
              COMPRESSED
            DUPLICATES ARE 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
    MEMBER IS JOB
        INDEX DBKEY POSITION IS AUTO
        OPTIONAL AUTOMATIC
        ASCENDING KEY IS ( JOB-TITLE ) UNCOMPRESSED
            DUPLICATES ARE NOT ALLOWED
    .
 ADD
SET NAME IS MANAGES
    ORDER IS SORTED
    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
     ASCENDING KEY IS ( WKRK-EMP-ID ) UNCOMPRESSED
            DUPLICATES ARE NOT ALLOWED
    .
 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 EMP-FIRST-NAME )
              COMPRESSED
            DUPLICATES ARE LAST
    .
 ADD
SET NAME IS REPORTS-TO
    ORDER IS SORTED
    MODE IS INDEX BLOCK CONTAINS 30 KEYS
    OWNER IS EMPLOYEE
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
    MEMBER IS STRUCTURE
        INDEX DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
        LINKED TO OWNER
            OWNER DBKEY POSITION IS AUTO
        OPTIONAL MANUAL
    ASCENDING KEY IS ( SUPR-EMP-ID ) UNCOMPRESSED
        DUPLICATES ARE FIRST
    .
 ADD
SET NAME IS EMP-PROJECT
    ORDER IS NEXT
    MODE IS CHAIN LINKED TO PRIOR
    OWNER IS EMPLOYEE
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
    MEMBER IS PROJECT
        INDEX DBKEY POSITION IS AUTO
        LINKED TO OWNER
            OWNER DBKEY POSITION IS AUTO
        OPTIONAL AUTOMATIC
    .
 ADD
SET NAME IS PROJECT-WORKER
    ORDER IS NEXT
    MODE IS CHAIN LINKED TO PRIOR
    OWNER IS PROJECT
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
    MEMBER IS WORKER
        INDEX DBKEY POSITION IS AUTO
        LINKED TO OWNER
            OWNER DBKEY POSITION IS AUTO
```

```
        OPTIONAL MANUAL
    .

ADD
SET NAME IS EMP-WORKER
    ORDER IS FIRST
    MODE IS CHAIN LINKED TO PRIOR
    OWNER IS EMPLOYEE
        NEXT DBKEY POSITION IS AUTO
        PRIOR DBKEY POSITION IS AUTO
    MEMBER IS WORKER
        INDEX DBKEY POSITION IS AUTO
        LINKED TO OWNER
            OWNER DBKEY POSITION IS AUTO
        MANDATORY AUTOMATIC
    .
 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 ) UNCOMPRESSED
            DUPLICATES ARE FIRST
    .
```

Appendix C: Zoned and Packed Decimal Fields as IDMS Keys

This section contains the following topics:

Overview

In many scenarios, it is necessary to construct IDMS keys that contain numeric fields with a format of zoned or packed decimal. To ensure the proper logical results intended by the database designer, you should be aware of the manner in which IDMS handles different variations of these fields and how application coding may influence the resulting contents of the database.

Numeric Formats

To understand the various ramifications of using zoned or packed decimal fields as IDMS key fields, it is necessary to have an understanding of the internal structure of the various formats.

Zoned decimal fields use one byte of storage to represent each single digit within a value. The high-order nibble of the last byte is used to convey the sign of the number when the field is defined as 'signed'. When a value is moved into an unsigned field, the sign nibble always contains a hexadecimal 'F'. A field that is signed uses a 'C' for a positive value and a 'D' for a negative number. It should be noted that a signed field also interprets an 'F' as a positive sign and a 'B' to represent a negative number. The high-order nibbles of all other bytes will contain a hex 'F' and are ignored for determining the sign of the number. The values of +999 and -999 will have the following internal structures when zoned decimal format is used.

```
Signed:       PIC S9(4)     +999  =  x'F0F9F9C9'
                                -999  =  x'F0F9F9D9'

Unsigned:  PIC  9(4)        +999  =  x'F0F9F9F9'
                                -999  =  x'F0F9F9F9'
```

Packed decimal format uses a single nibble for each digit of the number and maintains the sign in the low-order nibble of the field's last byte. Unsigned fields always use an 'F' for the sign while signed fields use a 'C' for positive numbers and a 'D' for negative numbers. Signed fields interpret an 'F' as a positive sign and a 'B' as a negative sign. The following internal structures will result for values of +999 and -999 when packed decimal format is used.

```
Signed:       S9(5) COMP-3        +999  =  x'00999C'
                                          -999  =  x'00999D'

Unsigned:      9(5) COMP-3        +999  =  x'00999F'
                                          -999  =   x'00999F'
```

It is important to realize that the various sign nibble values are assigned in a language such as COBOL when a value is moved directly into a named field. Fields that are part of group moves will not have any conversion performed relative to the value in their sign nibble.

```
02   GROUP-A.
  04    FIELD-A        PIC  S9(5) COMP-3.

02   GROUP-B.
  04    FIELD-B        PIC    9(5) COMP-3.
```

A value of -999 will be moved into FIELD-A and FIELD-B with the following instructions resulting in the hex value to the right of the instruction.

```
MOVE -999 TO FIELD-A.              FIELD-A = x'00999D'
MOVE -999 TO FIELD-B.              FIELD-B = x'00999F'
```

Although functionally equivalent the following instructions will result in a different value to be moved into FIELD-B.

```
MOVE -999 TO FIELD-A               FIELD-A = x'00999D'
MOVE GROUP-A to GROUP-B.       FIELD-B = x'00999D'
```

Although the above example used packed decimal numbers, the same scenario is true for zone decimal fields. This programming difference may have an impact on your IDMS database depending on the type of key in which a field is used and whether a field's definition is signed or unsigned.

Signed Versus Unsigned Keys

The most significant difference when using numeric fields as part of an IDMS key is whether the field has been defined as signed or unsigned. If a field has been defined as signed whether zoned or packed, IDMS will honor the format and will perform comparisons that will recognize functionally equivalent values as being equal.

```
x'00999C'  equals  x'00999F'
```

However unsigned fields are treated as character values and functionally equivalent values are not considered equal unless the sign nibbles are also equal.

```
X'00999C'  is not equal to  x'00999F'
```

Sorted Chain or Index Sets

The format of a field that is part of a key is significant to IDMS when that field is specifically named as part of the key. If a numeric field is part of a group level element and the group name is specified as the key, IDMS is not aware of the elementary elements at run time and the entire group is treated as character format. No numeric format specific comparisons are attempted by IDMS against elementary elements that are a part of a group when the group name is specified as the key field.

When signed zoned decimal or signed packed decimal fields are identified as part of a set or index key, IDMS honors the format. At run-time the DBMS will normalize the sign nibbles so that comparisons against functionally equivalent values return a result of equal. This ensures that sorted chain sets and index sets will maintain their sequence based on a field's functional value.

Since unsigned zone decimal or packed decimal numbers are treated as character data, functionally equivalent values with different sign nibbles return a result of not equal during comparison operations. This can result in a different sequence for a sorted set or index set depending on the sign characteristic of the numeric field as depicted in the following example. Each set is assumed to have an ascending order and allows duplicates.

```
Signed:                                      Unsigned:
 02  FIELD-A   PIC  S9(5)  COMP-3.   02  FIELD-A   PIC  9(5)  COMP-3.
 02  FIELD-B   PIC  x(4).                 02  FIELD-B    PIC  X(4).

    x'00999C', c'AAAA'                    x'00999C', c'AAAA'
    x'00999F', c'AAAA'                    x'00999C', c'BBBB'
    x'00999C', c'BBBB'                    x'00999F', c'AAAA'
```

In the signed example, x'00999C' and x'00999F' are functionally equal and the relative position of the first two records is determined by the duplicates option and the sequence in which the records were added to the set. For the unsigned example the functional equivalence of the packed fields will not be recognized and x'00999F' is considered to be greater than x'00999C'. The duplicates option would not come into consideration since none of the three keys is considered to be equal.

CALC Records

Inconsistent sign nibbles for zone or packed decimal fields used in a calckey may have more of an impact than when those fields are used as keys for a chain set or index set. The initial operation applied against a CALC record runs a hashing algorithm against the calckey to identify the page on which IDMS will store the record occurrence. Prior to executing the hashing algorithm, IDMS constructs the calckey into a piece of contiguous storage from the various fields defined as making up the key. The algorithm has no knowledge of the individual component fields comprising the calckey or their various formats. The algorithm simply performs logical arithmetic calculations against the character string to determine the record's target page.

As a result, values of x'00999C' and x'00999F' although functionally equivalent, will in all probability generate different target pages for their CALC records. If a zone or packed field is used as part of a calckey it is very important that all programs involved with the creation or accessing of these records use a consistent method for initializing these numeric fields.

Index

O

P

R

S

T

U

V

W