CA Datacom®/DB

SQL User Guide for z/VSE

Version 12.0, First Edition

CA Technologies Product References

This document references the following CA products:

- CA Datacom®/DB
- CA Datacom® CICS Services
- CA Datacom® Datadictionary™
- CA Datacom® DB2 Transparency
- CA Datacom® DL1 Transparency
- CA Datacom® IMS/DC Services
- CA Datacom® Server
- CA Datacom® SQL (SQL)
- CA Datacom® STAR
- CA Datacom® TOTAL Transparency
- CA Datacom® VSAM Transparency
- CA Dataquery™ for CA Datacom® (CA Dataquery)
- CA Ideal™ for CA Datacom® (CA Ideal)
- CA IPC
- CA Librarian®
- CA Common Services for z/OS

Contact CA Technologies

Contact CA Support

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

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

Providing Feedback About Product Documentation

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

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

Contents

Chapter 1: Introduction 9

System Tasks 9
Syntax Diagrams 9
CA Datacom/DB Extensions 9
Where to Find Information 9
Related Publications 9
Listing Libraries for CA Datacom Products 10
Reading Syntax Diagrams 11
- Statement Without Parameters 11
- Statement with Required Parameters 12
- Delimiters Around Parameters 12
- Choice of Required Parameters 12
- Default Value for a Required Parameter 13
- Optional Parameter 13
- Choice of Optional Parameters 13
- Repeatable Variable Parameter 14
- Separator with Repeatable Variable and Delimiter 14
- Optional Repeatable Parameters 15
- Default Value for a Parameter 15
- Variables Representing Several Parameters 15

CA Datacom/DB Extensions 16

Chapter 2: Before You Start 17

What Is SQL? 17
What You Should Know About SQL 17
- Tables 17
- Columns 17
- Rows 18
- Views 18
- Table and View Examples 18
- Indexes 20
- Cursors 20
- Units of Work 20
- Units of Recovery 20
- Isolation Levels 21
- Schemas 23

Authorization ID 23
Accessor ID 23
Privileges 24
Synonym 24
SQL Statements 25
Binding 27
Plan 27
SQL Manager 28
Reserved Words 29

Chapter 3: Getting Started 33

SQL Schemas 33
SQL Tables 33
SQL Tables and Logging 34
Creating SQL Tables 34
Using Existing Tables 35
Populating SQL Tables 36
Accessing SQL Tables 37
Selecting and Manipulating Data 37
Specifying Preprocessor Options 38
Preparing Programs 38
Mixed Mode Programming 39
Statement Execution Table 39
Dynamic SQL 41
Static SQL 42
Dynamic SQL 42
Dynamic SQL in CA Datacom/DB 43
INCLUDE Directive 43
Name Types 44
Reserved Words 44
Parameter Markers 44
Security Implications of Dynamic SQL 44
Using Dynamic SQL in Application Programs 44
Other Tasks 55
SQL Status Tables 56
Procedures and Triggers 56
Overview 57
SQL Procedures 60
External Security Support for Procedure/Trigger Creation and Execution 60
Trigger Execution for Record-at-a-Time Maintenance 61
Transaction Integrity 61

Subroutine Calls Inside Procedures 62
Restrictions 62
Multi-User Facility Considerations for Procedures 63
Parameter Styles and Error Handling 66
SQL Error Messages Related to Procedures and Triggers 69
Datadictionary Support for Triggers and Procedures 72
Examples: Creating a Procedure 73
Example: Calling a Procedure 92
Left Outer Joins 93
Overview of Joins 93
SELECT Statement Subselect Syntax 94
SELECT Statement Select-Into Syntax 94
Inner Join Example 95
Outer Join Example 95
Value of Rows That Do Not Match 95
WHERE Clause 96
Performance Considerations 97
Order of Predicate Evaluation 98
SQL Memory Guard 99
Activating the SQL Memory Guard 100
SQL and Multiple Multi-User Facilities Support 101
Application Design Considerations 101
Index-Only Processing 102
Cursor Processing 102
DBSQLPR 103
Processing 104
Line Commands 104
DBSQLPR Syntax 105
DBSQLPR Options 105
DROP PLAN (DBSQLPR) 114
Example JCL 115
Sample Report 117
DATACOM VIEWs 118
Overview 118
Redefinitions 118
Arrays 121
Default Values for Redefinitions and Arrays 122

Chapter 1: Introduction

This guide tells you how to use SQL with CA Datacom/DB.

This guide is intended for those who write application programs with embedded SQL statements and create/maintain personal SQL tables. It does not attempt to address details of coding programs with CA Datacom/DB or CA Datacom Datadictionary commands. These can be found in the *CA Datacom/DB Programming Guide* and the *CA Datacom Datadictionary DSF Programming Guide*.

System Tasks

Datadictionary Administrators, Database Administrators, and systems programmers can find SQL information pertinent to their system tasks in the *CA Datacom/DB Database and System Administration Guide*.

Syntax Diagrams

For information on how to read the syntax diagrams in this guide, see Reading Syntax Diagrams (see page 11).

CA Datacom/DB Extensions

For information on CA Datacom/DB extensions to ANSI standard SQL implementations, see CA Datacom/DB Extensions (see page 16).

Where to Find Information

See the index to quickly locate information on specific subjects.

Related Publications

In addition to this guide, you need the *CA Datacom/DB Message Reference Guide*.

Listing Libraries for CA Datacom Products

Guidelines to assist you in preparing your JCL are provided in this manual. The sample code provided in this document is intended for use as a reference aid only and no warranty of any kind is made as to completeness or correctness for your specific installation.

Samples for JCL and programs are provided in the install library. In in z/OS, the default name for this library is product specific. For example, the macro library for DB is CABDMAC. In z/VSE, sample PROCs are provided that allow you to make use of parameter substitution. You can copy and modify these samples for your specific requirements.

Any JOB statements should be coded to your site standards and specifications. All data set names and library names should be specified with the correct names for the installation at your site. In many examples, a REGION= or SIZE= parameter is displayed in an EXEC statement. The value displayed should be adequate in most instances, but you can adjust the value to your specific needs.

The libraries listed for searching must include the following in the order shown:

1. User libraries (*hlq*.CUSLIB)you may have defined for specially assembled and linked tables, such as DBMSTLST, DBSIDPR, DDSRTLM, DQSYSTBL, or User Requirements Tables
2. CA Datacom base libraries (*hlq*.CABDLOAD): CA Datacom/DB, CA Datacom Datadictionary, CA Dataquery, SQL
3. CA IPC libraries (*hlq*.CAVQLOAD)
4. CA Common Services (formerly known as CA Common Services for z/OS) base libraries (*hlq*.CAW0LOAD)
5. Libraries for additional products, such as CA Datacom CICS Services, CA Datacom VSAM Transparency, CA Ideal, and so on

CA Dataquery users also need the following libraries and data sets for the following specific functions:

- The z/OS data set DQOUT or the z/VSE data set DQOUTD is used only if the DQBATCH execution uses the EXPORT function.
- In z/OS, running deferred queries with separate JCL members in batch requires, in addition to the SYSIN statement DEFER, the inclusion of a DD statement for the internal reader used by VPE. This DD statement should be:

```
//IRDR DD     SYSOUT=(A,INTRDR)
```

Reading Syntax Diagrams

Syntax diagrams are used to illustrate the format of statements and some basic language elements. Read syntax diagrams from left to right and top to bottom.

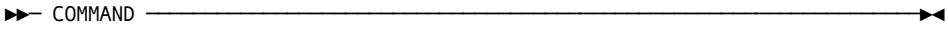
The following terminology, symbols, and concepts are used in syntax diagrams:

- Keywords appear in uppercase letters, for example, COMMAND or PARM. These words must be entered exactly as shown.
- Variables appear in italicized lowercase letters, for example, *variable*.
- Required keywords and variables appear on a main line.
- Optional keywords and variables appear below a main line.
- Default keywords and variables appear above a main line.
- Double arrowheads pointing to the right indicate the beginning of a statement.
- Double arrowheads pointing to each other indicate the end of a statement.
- Single arrowheads pointing to the right indicate a portion of a statement, or that the statement continues in another diagram.
- Punctuation marks or arithmetic symbols that are shown with a keyword or variable must be entered as part of the statement or command. Punctuation marks and arithmetic symbols can include the following:

,	comma	>	greater than symbol
.	period	<-	less than symbol
(	open parenthesis	=	equal sign
)	close parenthesis	¬	not sign
+	addition	-	subtraction
*	multiplication	/	division

Statement Without Parameters

The following is a diagram of a statement without parameters:



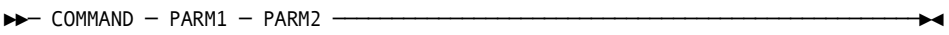
For this statement, you must write the following:

```
COMMAND
```

Statement with Required Parameters

Required parameters appear on the same horizontal line, the main path of the diagram, as the command or statement. The parameters must be separated by one or more blanks.

The following is a diagram of a statement with required parameters:



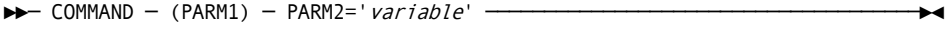
You must write the following:

```
COMMAND PARM1 PARM2
```

Delimiters Around Parameters

Delimiters, such as parentheses, around parameters or clauses must be included.

The following is a diagram of a statement with delimiters around parameters:



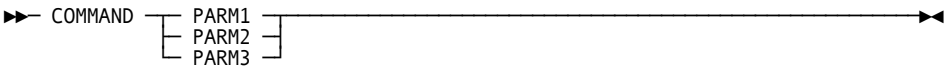
If the word *variable* is a valid entry, you must write the following:

```
COMMAND (PARM1) PARM2='variable'
```

Choice of Required Parameters

When you see a vertical list of parameters as shown in the following example, you must choose one of the parameters. This indicates that one entry is required, and only one of the displayed parameters is allowed in the statement.

The following is a diagram of a statement with a choice of required parameters:



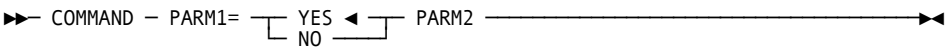
You can choose one of the parameters from the vertical list, such as in the following examples:

```
COMMAND PARM1
COMMAND PARM2
COMMAND PARM3
```

Default Value for a Required Parameter

When a required parameter in a syntax diagram has a default value, the default value appears above the main line, and it indicates the value for the parameter if the command is not specified. If you specify the command, you must code the parameter and specify one of the displayed values.

The following is a diagram of a statement with a default value for a required parameter:



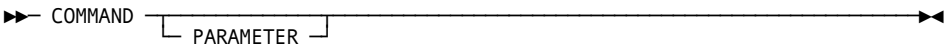
If you specify the command, you must write one of the following:

```
COMMAND PARM1=NO PARM2
COMMAND PARM1=YES PARM2
```

Optional Parameter

A single optional parameter appears below the horizontal line that marks the main path.

The following is a diagram of a statement with an optional parameter:



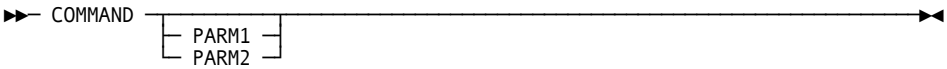
You can choose (or not) to use the optional parameter, as shown in the following examples:

```
COMMAND
COMMAND PARAMETER
```

Choice of Optional Parameters

If you have a choice of more than one optional parameter, the parameters appear in a vertical list below the main path.

The following is a diagram of a statement with a choice of optional parameters:



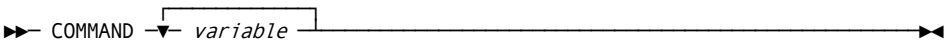
You can choose any of the parameters from the vertical list, or you can write the statement without an optional parameter, such as in the following examples:

```
COMMAND
COMMAND PARM1
COMMAND PARM2
```

Repeatable Variable Parameter

In some statements, you can specify a single parameter more than once. A repeat symbol indicates that you can specify multiple parameters.

The following is a diagram of a statement with a repeatable variable parameter:



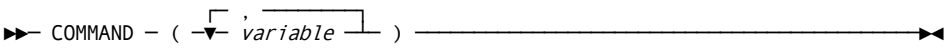
In the preceding diagram, the word *variable* is in lowercase italics, indicating that it is a value you supply, but it is also on the main path, which means that you are required to specify at least one entry. The repeat symbol indicates that you can specify a parameter more than once. Assume that you have three values named VALUEX, VALUEY, and VALUEZ for the variable. The following are some of the statements you might write:

```
COMMAND VALUEX
COMMAND VALUEX VALUEY
COMMAND VALUEX VALUEX VALUEZ
```

Separator with Repeatable Variable and Delimiter

If the repeat symbol contains punctuation such as a comma, you must separate multiple parameters with the punctuation. The following diagram includes the repeat symbol, a comma, and parentheses:

The following is a diagram of a statement with a separator with a repeatable variable and a delimiter:



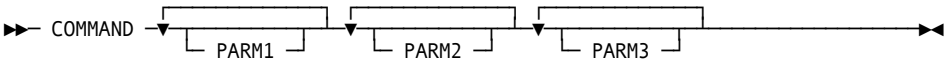
In the preceding diagram, the word *variable* is in lowercase italics, indicating that it is a value you supply. It is also on the main path, which means that you must specify at least one entry. The repeat symbol indicates that you can specify more than one variable and that you must separate the entries with commas. The parentheses indicate that the group of entries must be enclosed within parentheses. Assume that you have three values named VALUEA, VALUEB, and VALUEC for the variable.

The following are some of the statements you can write:

```
COMMAND (VALUEC)
COMMAND (VALUEB,VALUEC)
COMMAND (VALUEB,VALUEA)
COMMAND (VALUEA,VALUEB,VALUEC)
```

Optional Repeatable Parameters

The following diagram shows a list of parameters with the repeat symbol for optional repeatable parameters:



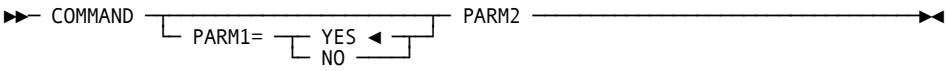
The following are some of the statements you can write:

```
COMMAND PARM1
COMMAND PARM1 PARM2 PARM3
COMMAND PARM1 PARM1 PARM3
```

Default Value for a Parameter

The placement of YES in the following diagram indicates that it is the default value for the parameter. If you do not include the parameter when you write the statement, the result is the same as if you had actually specified the parameter with the default value.

The following is a diagram of a statement with a default value for an optional parameter:

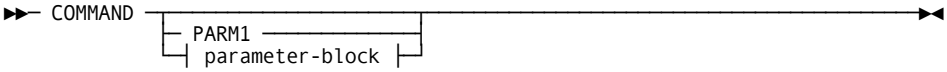


For this command, COMMAND PARM2 is the equivalent of COMMAND PARM1=YES PARM2.

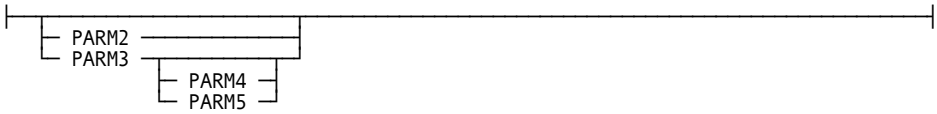
Variables Representing Several Parameters

In some syntax diagrams, a set of several parameters is represented by a single reference.

The following is a diagram of a statement with variables representing several parameters:



Expansion of parameter-block



The *parameter-block* can be displayed in a separate syntax diagram.

Choices you can make from this syntax diagram therefore include, but are not limited to, the following:

```
COMMAND PARM1
COMMAND PARM3
COMMAND PARM3 PARM4
```

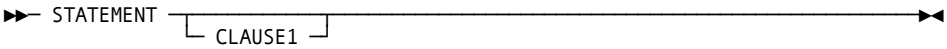
Note: Before you can specify PARM4 or PARM5 in this command, youmust specify PARM3.

CA Datacom/DB Extensions

The CA Datacom/DB implementation of SQL conforms to American National Standard Database Language SQL, ANSI X3.135-1989 in the following manner:

1. Full SQL conformance to level 2, except as noted.
2. Includes implementation of the following facilities:
   a. Direct processing of SQL data manipulation language statements.
   b. Embedded SQL COBOL and PL/I.

In some instances, the SQL implementation described in this manual has added capabilities beyond the SQL standard. CA Datacom/DB extensions to ANSI standard SQL are indicated in the label of the syntax diagram's box (if the entire diagram represents a CA Datacom/DB extension), or within the diagram as shown below (if only part of a diagram represents a CA Datacom/DB extension).



Note: CLAUSE1 is a CA Datacom/DB extension.

You can choose to use the CA Datacom/DB extension, which is essentially an optional parameter. Some of these extensions are unique to the CA Datacom/DB environment and could function differently from syntactically similar extensions provided by other implementations of SQL.

Chapter 2: Before You Start

What Is SQL?

SQL is a database sub-language which you can use to define, manipulate and control data in your relational databases. As part of our ongoing commitment to protect our clients' investments in application software resources, CA Datacom/DB offers SQL support as a fully integrated part of CA Datacom/DB. We intend CA Datacom/DB SQL to provide support that offers a broad scope of facilities for the development of applications while minimizing the amount of effort required to port those applications from one DBMS to another.

SQL allows you to perform powerful relational functions such as projection, restriction, joining and union.

In performing tasks using SQL, you can draw on support provided by other DATACOM products such as CA Datacom Datadictionary and CA Dataquery.

What You Should Know About SQL

While a database must satisfy many requirements to be classified as a relational database, one of the requirements is that the data appears to you as a collection of tables.

Tables

SQL allows you to access tables as sets of data. A base table is the table as it is defined and contained in the database. You can form result tables by accessing only part of the data stored in a base table. Each table consists of a specific number of columns and an unordered collection of rows.

Columns

Columns are the vertical components of the table. A column describes an indivisible unit of data. Each column has a name and a particular data type, such as character or integer. While the order of columns in a table is fixed, there is no conceptual significance to this order.

Rows

The horizontal components of tables are called rows. A row is a sequence of values, one for each column of the table. Each row contains the same number of columns. You insert and delete rows, whereas you update individual columns. A table, by the way, can exist without any rows.

Views

Using SQL, you can define views, which are alternative representations of data from one or more tables.

A view is a derived table or a subset of the columns and rows of the table on which it is defined. A view can also be defined on another view.

The capability of joining two or more tables easily is a major advantage that distinguishes relational systems from nonrelational systems. The ability to create views, or derived tables, allows you to access and manipulate only that data which is significant for your purposes.

Table and View Examples

Following is a conceptual diagram of a table named PERSONNEL:

	EMPNO	LNAME	FNAME	MI	CITY	ST
ROW 1	010900	Duparis	Jean	C	Houston	TX
ROW 2	008206	Santana	Juan	M	Dallas	TX
ROW 3	002105	MacBond	Sean	D	El Paso	TX
ROW 4	010043	Odinsson	Jon	L	Dallas	TX

Following is a conceptual diagram of a table named PAY:

	EMPNO	SALARY	YTDCOM
ROW 1	010043	03560000	00120000
ROW 2	008206	04530000	00290000
ROW 3	010900	02970000	00075000
ROW 4	002105	03280000	00107500

The two previously shown tables contain information about the same four people (match the EMPNO columns), but the order of the rows in each table is not significant.

However, the columns appear in the same order in each row. For example, in the PERSONNEL table, EMPNO is always first, LNAME is always second, FNAME is always third, and so on.

The values which appear in a column fall within the same type, that is to say, LNAME, FNAME and MI each contain character data, while SALARY contains numeric data.

The values which appear in LNAME all fall within the range of valid values, or domain, of "last name," the values in FNAME are within the domain of "first name," and the values in SALARY are within the domain of "salary" which is $999,999.99 to 0.00 for this example.

Some columns, such as ST (for "state"), may contain duplicate values (in this case, TX), but that does not mean that TX is the only value in the domain for the column ST.

Other columns contain only unique values, such as EMPNO, since no two employees of this company have the same employee number. In the previous example, the employee number is used to uniquely identify information about each employee no matter which table contains the information.

Using the tables in the previous example, you could define a view that allows you to see the name of each employee (columns LNAME, FNAME and MI from the PERSONNEL table) and the salary of that employee (column SALARY from the PAY table). In your "view," the information you requested would look like a table and actually joins specified data from two different tables.

Following is a conceptual diagram of a view which you have named WAGES:

	LNAME	FNAME	MI	SALARY
ROW 1	Duparis	Jean	C	02970000
ROW 2	Santana	Juan	M	04530000
ROW 3	MacBond	Sean	D	03280000
ROW 4	Odinsson	Jon	L	03560000

The view WAGES, derived from the tables PERSONNEL and PAY, thus shows a "view" of only the columns that you want to see.

Indexes

Tables are often accessed by the data values contained in one or more columns. To make such accesses efficient, the tables can be indexed by one or more columns. Such an index supports direct access to the table's rows by their data value content. A given table can support multiple indexes. CA Datacom/DB automatically maintains the index as the table's content changes.

Indexes are a performance-only consideration for you, the SQL user. The presence or absence of an index does not enhance or restrict the logical operations supported for a table. CA Datacom/DB also supports a special type of index to control the physical placement of rows to enhance performance. This "clustering index" is automatically the lowest level, no locks are acquired for rows accessed "read only" created by the system if your Database Administrator has selected this space management option.

Cursors

You can control the row to which an application program points by manipulating a control structure called the cursor. You can use the cursor to retrieve rows from an ordered set of rows, possibly for the purpose of updating or deleting. The SQL statements FETCH, UPDATE, and DELETE support the concept of positioned operations.

Units of Work

A unit of work contains one or more units of recovery. In a batch environment, a unit of work corresponds to the execution of an application program. Within that program, there may be many units of recovery as COMMIT or ROLLBACK statements are executed.

Units of Recovery

A unit of recovery is a sequence of operations within a unit of work and includes the data and control information needed to enable CA Datacom/DB to back out or reapply all of an application's changes to recoverable resources since the last commit point. A unit of recovery is initiated when a unit of work starts or by the termination of a previous unit of recovery. A unit of recovery is terminated by a commit or rollback operation or the termination of a unit of work. The commit or rollback operation affects only the results of SQL statements and CA Datacom/DB commands executed within a single unit of recovery.

Isolation Levels

Units of recovery can be isolated from the updating operations of other units of recovery. This is called isolation level.

The "uncommitted data" isolation level allows you to access rows that have been updated by another unit of recovery, but the changes have not been committed, or written to the base table.

The isolation level that provides a higher degree of integrity is the "cursor stability" isolation level. With cursor stability, a unit of recovery holds locks only on its uncommitted changes and the current row of each of its cursors.

The "repeatable read" isolation level provides maximum protection from other executing application programs. When your program executes with repeatable read protection, rows referenced by your program cannot be changed by other programs until your program reaches a commit point.

Note: In a Data Sharing environment, an isolation level of repeatable read is not supported across the MUFplex. For more information on Data Sharing, see the *CA Datacom/DB Database and System Administration Guide*.

Repeatable Read Interlocks

The repeatable read transaction isolation level provides the highest level of isolation between transactions because it acquires a share or exclusive *scan range intent lock* before beginning a scan (all rows are accessed with the scan operation). This lock is released when the transaction ends, guaranteeing that other transactions cannot update, delete or insert rows within the scan range until the transaction ends. If another transaction attempts to do so, it waits until the transaction has ended, or one of the transactions is aborted if an exclusive control interlock occurs. As the name implies, a repeatable read transaction is therefore guaranteed to reread the exact same set of rows if it reopens a cursor or re-executes a SELECT INTO statement (any changes made by the transaction itself would of course be visible).

Although repeatable read isolation provides a convenient way to isolate transactions, it does so at the cost of possible lower throughput and more exclusive control interlocks, as described in the following:

- Lower Throughput:

  Because more rows remain locked for a longer period of time, repeatable read isolation may lower total throughput (transactions wait longer for locks to be released).

- Mixed Repeatable Read and Cursor Stability Transactions:

  Repeatable read may cause more exclusive control interlocks, especially if concurrent transactions are not using repeatable read. For example, if cursor stability transaction CS updates row R1, and then repeatable read transaction RR acquires a scan range intent lock that includes row R1, CS waits if it attempts to read a row in RR's scan range with exclusive control. While CS is waiting, unless row R1 has already been read by RR's scan, RR eventually attempts to read row R1 with a row-level share lock. But because CS is waiting on RR, neither transaction can continue. So, the deadlock condition is resolved by abnormally terminating RR, which releases its locks and allows CS to continue. In this case, if transaction CS is changed to repeatable read isolation, it acquires an exclusive scan range intent lock before updating row R1. Transaction RR then waits when it attempts to acquire its scan range intent lock.

- Scan Range May be Entire Table:

  A deadlock can still occur if concurrent repeatable read transactions acquire multiple scan range intent locks. The same conditions exist as with row-level locking of cursor stability transactions, except that with repeatable read a larger number of rows may be locked with the scan ranges, and these locks are held for a longer period of time. This is especially true when the first column of the scan index is not restricted, or multiple indexes are merged. In these cases, the scan range is the entire table.

- Avoiding Deadlocks:

  If deadlock avoidance is critical, it can be avoided if all concurrent transactions execute LOCK TABLE statements in the same sequence before executing any other statements in a transaction. If the transaction might insert, update or delete rows of a table, the lock must be exclusive, and this causes all other transactions attempting to execute a LOCK TABLE statement for the table, or tables, to wait. Because the LOCK TABLE statements are executed in the same sequence, perhaps by table name, no deadlock can occur.

Schemas

A schema is a collection of tables, views, synonyms, and plans which make up an SQL environment. Schemas may be created so that each user has a *personalized* SQL environment by creating a schema for each user. Schemas may also be created that reflect some other organization of data, such as by department or project. Or a combination of both approaches may be used.

Authorization ID

The name of a schema is known as its authorization ID. A fully-qualified table, view, synonym, or plan name consists of the name of the object and the authorization ID of the schema to which the object belongs. If an authorization ID is not explicitly specified, the default authorization ID in effect is assumed.

Note: For application programs, the default authorization ID is the one named in the AUTHID= Preprocessor option. For information about how the default authorization ID is specified in CA Datacom Datadictionary, see Relating the Person to the AUTHID. For information about how the default authorization ID is specified in CA Dataquery, see the *CA Dataquery User Guide*.

Accessor ID

An accessor ID designates a user. Note that this is a user's ID, not a schema's authorization ID.

Privileges

Security is typically handled using the CA Datacom/DB External Security Model. With external security, access rights to the underlying data are controlled through table, plans, or view rights, defined in the external security product.

Optionally, you may secure access using the SQL Security Model. With the SQL Security Model, privileges are automatically granted to the owner when a table or view is created. The owner may then grant and revoke those privileges to others by issuing GRANT and REVOKE statements. With external security, there is no automatic granting of privileges.

Note: Privileges in CA Datacom/DB are granted to users, not to schema IDs. For example, when a table is created the table is defined to be in a particular schema. But the privileges which are automatically granted are given to the accessor ID of the user who executed the CREATE TABLE statements. Similarly, when privileges are granted, they are granted to users, not schemas.

Synonym

Synonyms are alternative names for tables and views. The full name of a table or view is qualified by the authorization ID. You can avoid using the full name by defining a synonym for a specific table or view. These short names are especially useful if accessing a table or view owned by another schema.

SQL Statements

You embed SQL statements in a host program written in a host language such as COBOL or PL/I. Variables defined in the host program that are referenced by the SQL statements are called host variables.

You can also submit certain SQL statements through the CA Datacom Datadictionary Interactive SQL Service Facility or interactively through CA Dataquery. See the Statement Execution Table (see page 39).

CA Datacom/DB supports the dynamic preparation and execution of SQL statements under the control of an application program. See Dynamic SQL (see page 41).

The SQL sub-language consists of the following:

Data Definition Language (DDL)

DDL statements define the SQL objects, such as tables and views.

Note: Because DDL statements are not recorded to the Log Area (LXX), they are not recoverable using the RECOVERY function of the CA Datacom/DB Utility (DBUTLTY). In the case of DDL statements, it is therefore your responsibility to ensure the existence of the Directory (CXX) definitions necessary for recovery.

Data Manipulation Language (DML)

DML statements let you access and manipulate the data in your SQL tables.

Note: You cannot use SQL DML statements to do maintenance on the DATA-DICT database, that is, no maintenance can be done to any tables in the DATA-DICT database using SQL. For details about DATA-DICT, see the *CA Datacom/DB Database and System Administration Guide*.

SQL Control Statements

Includes the CALL and EXECUTE PROCEDURE statements that supports the implementation of procedures and triggers beginning in r10.

The following table lists the SQL statements in the categories of DDL, DML, and SQL Control Statements:

Data Definition Language (DDL)	Data Manipulation Language (DML)	SQL Control Statements
ALTER TABLE COMMENT ON CREATE INDEX CREATE PROCEDURE CREATE RULE CREATE SCHEMA CREATE SYNONYM CREATE TABLE CREATE TRIGGER CREATE VIEW DROP GRANT REVOKE	**Cursor operations:** CLOSE DECLARE CURSOR DELETE...CURRENT (positioned DELETE) FETCH OPEN UPDATE...CURRENT (positioned UPDATE) **Non-cursor operations:** DELETE (searched DELETE) INSERT SELECT UPDATE (searched UPDATE) **Exception handling operations:** WHENEVER	CALL EXECUTE PROCEDURE

The following table lists the dynamic SQL and SQL session statements:

Dynamic SQL Statements	SQL Session Statement
DESCRIBE dynamic DECLARE dynamic FETCH dynamic OPEN EXECUTE EXECUTE IMMEDIATE PREPARE	SET CURRENT SQLID

See the descriptions of the SQL statements beginning with ALTER TABLE for information on how to use these statements.

Binding

SQL statements must be prepared during the program preparation process before the program is executed. This process is called binding. The SQL Preprocessor prepares the SQL portions of a source program for execution.

CA Datacom/DB delays some decisions which impact the method used to execute an SQL statement until execution time if information required to make the best decision is not available until execution time. This technique is called phased binding. In effect, the binding process is performed in discrete phases and one of those phases does not occur until execution time.

For SQL statements embedded in a host language, such as COBOL, binding is performed when the program is preprocessed. For SQL statements executed through CA Dataquery, binding occurs during the validation step. For the CA Datacom Datadictionary, binding occurs automatically when SQL statements are executed.

When a statement is prepared, any dependencies of that statement on table or view definitions are recorded in the CA Datacom Datadictionary. If any dependent objects are changed, the related statement is marked invalid and must be rebound before it can be executed again.

The SQL Manager automatically attempts a rebind when an invalid statement is executed. Rebinding can also be requested in advance. For more information, see *CA Datacom/DB SQL Preprocessors*.

Plan

A product of the binding process is the CA Datacom/DB access plan. The plan is required by CA Datacom/DB to process SQL statements encountered during execution. The preparation phase builds the plan for the application and binds a statement to table, view and synonym definitions stored in the CA Datacom Datadictionary. This eliminates the cost of binding at each execution of a statement.

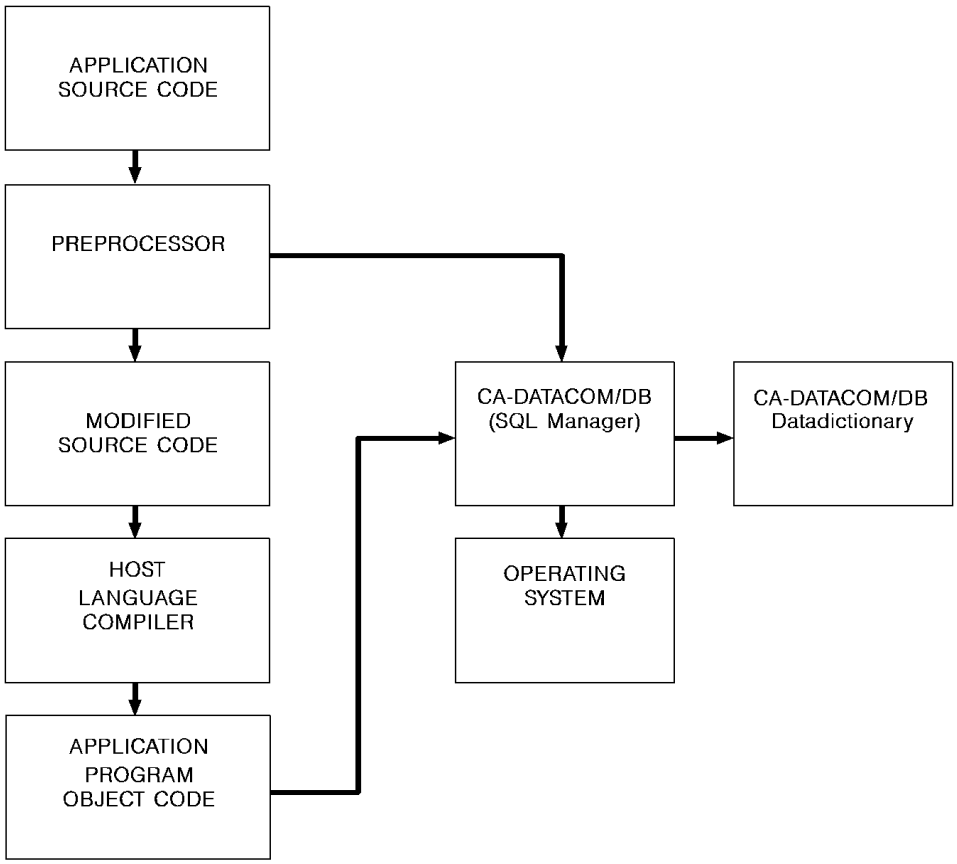
Since SQL plans are stored in the CA Datacom Datadictionary, the CA Datacom Datadictionary must be available to execute previously prepared SQL statements.

SQL plans are securable. With plan security you can create a plan such that, in order to execute the plan, an accessor ID must have the plan EXECUTE privilege for that plan. The plan EXECUTE privilege can be granted with the GRANT statement and revoked with the REVOKE statement. For other plan security information in this guide, see GRANT and REVOKE, and CHECKPLAN=, CHECKWHEN=, CHECKWHO=, and SAVEPLANSEC= options in Description of Options.

Note: For detailed information about plan security, see the *CA Datacom Security Reference Guide*.

SQL Manager

The SQL Manager prepares, optionally stores, and executes SQL statements. The SQL Manager is integrated with the Multi-User Facility and is accessed by the CA Datacom/DB SQL Preprocessor and by other DATACOM products such as the CA Datacom Datadictionary and CA Dataquery. The following diagram approximates how the SQL Manager processes an application with embedded SQL statements.



Interactive SQL is supported by CA Dataquery, with capabilities to create and populate tables using standard SQL statements. See the CA Dataquery documentation for information on how this product uses SQL.

The CA Datacom Datadictionaryy supports the definition of tables and views using SQL statements, allowing you to take advantage of the standardization that SQL provides. You can also use the CA Datacom Datadictionary menu-driven method to define database structures, as an alternative to SQL, and update the SQL defined databases with additional information that is not currently available in SQL. In addition, the CA Datacom Datadictionary provides many capabilities not available through SQL. For example, text classifications allow you to store text about your SQL tables, views and columns in addition to that specified in the COMMENT ON statement. Certain attributes for SQL tables and columns can be modified directly through CA Datacom Datadictionary without limiting the ability to access these occurrences through SQL.

Note: For more information about these capabilities, see the *CA Datacom Datadictionary Online Reference Guide*.

Reserved Words

The following table lists SQL reserved words. Do not form names using any of these words as SQL identifiers. See Identifiers.

Note: The SQL transport utility (DDTRSLM) has additional restrictions on words used for an AUTHID, SQL name, or CA Datacom Datadictionary occurrence name. See the *CA Datacom Datadictionary Batch Reference Guide*.

In the following table, where more than one word is listed on a line, those words as a group are reserved, not necessarily the individual words that make up the group unless that word is listed separately on its own line. Words followed by an asterisk (*) indicate that word is reserved only in the COBOL language. We reserve the right to add or change reserved keywords as needed.

SQL Reserved Words

ADD	BEFORE	CALL	DATA
AFTER	BEGIN	CASE	DATACOM
ALL	BETWEEN	CAST	DATACOM DUMP
ALTER	BIT	COALESCE	DATACOM LOOPLIMIT
AND	BIT_ADD	COLUMN	DATACOM TSN
ANY	BIT_AND	COBOL	DELETE
ARRAY	BIT_NOT	CONCAT	DESCRIPTOR
AS	BIT_OR	CONDITION	DETERMINISTIC
ASSEMBLER	BIT_XOR	CONTINUE HANDLER	DISTINCT
ASENSITIVE	BY	CONTAINS	DO
ATOMIC	BYREF	CONVERSION	DROP
		COUNT	
		CURRENT	
		CURSOR	
EACH	FIRST	GENERAL	HANDLER
ELSE	FOR	GET CURRENT DIAGNOSTICS	HAVING
ELSEIF	FROM	GET DIAGNOSTICS	
END		GET STACKED	
END-EXEC*		GET STACKED DIAGNOSTICS	
EXECUTE		GRANT	
EXISTS		GROUP	
EXIT HANDLER			
EXTERNAL			
IF	JOIN	KEY	LANGUAGE
IMMEDIATE			LEADING
IN			LEAVE
INDEX			LEFT
INNER			LIKE
INOUT			LOOP
INPUT			LOWER
INSENSITIVE			LOWERCASE
INSERT			LTRIM
ITERATE			
INTEXTRACT			
INTO			
INVALIDATE			
IS			

MODIFIES	NEW	OF	PARAMETER
MUF_NAME	NEWFUN1	OLD	PLI
	NEWFUN2	ON	PRIVILEGES
	NEWFUN3	OPTIMIZE	PROCEDURE
	NO	OPTION	PROGRAM
	NOT	OPTIONS	
	NOT FOUND	OR	
	NULL	ORDER	
	NULLIF	OUT	
	NULLS	OUTER	
RAISE ERROR	SELECT	TABLE	UNION
READS	SENSITIVE	THEN	UNDO HANDLER
REFERENCING	SET	TO	UNTIL
REPEAT	SIGNAL	TO PXXSQL	UPDATE
RESIGNAL	SOME	TO SYSOUT	UPPER
RETURN	SPECIFIC	TRAILING	UPPERCASE
RTRIM	SQL	TRIGGER	USER
RULE	SQLEXCEPTION	TRIM	USING
RUN	SQLSTATE	TSN	
	SQLWARNING		
	SQUEEZE		
	STATEMENT		
	STRIP		
	STYLE		
	SUBSTRING		
	SYNONYM		
VALUES	WHEN	XMLATTRIBUTES	
VARCHAR	WHERE	XMLCONCAT	
VIEW	WHILE	XMLELEMENT	
	WITH	XMLFOREST	
	WITHOUT	XMLSERIALIZE	

Chapter 3: Getting Started

SQL Schemas

Before you can use the capabilities SQL offers for defining, manipulating and controlling data, you must have a schema to define your SQL environment. A schema is required before you can use SQL.

The schema is essentially an authorization ID and all the SQL objects (tables, views, plans, and synonyms) qualified by that authorization ID.

You can create a schema in the following ways:

- Embed a CREATE SCHEMA statement in an application
- Submit the CREATE SCHEMA statement through the CA Datacom Datadictionary Interactive SQL Service Facility
- Use DBSQLPR to CREATE SCHEMA

When you create your schema, the only requirement is that you specify the authorization ID. You can optionally define your tables, views and synonyms at that time, or you can create these objects separately as needed.

SQL Tables

Once you have a schema, you must have tables that SQL can access. In the CA Datacom/DB environment, this means you must have a table definition in the CA Datacom Datadictionary, and it must be associated with a data area where the table data is to be stored. The data area must be defined in the CA Datacom Datadictionary and be associated with a specific database, which has been cataloged to the CA Datacom/DB Directory (CXX). You can use existing tables, if they meet the requirements for access by SQL. You can also modify existing tables so SQL can access them, or you can create your own tables.

Note: When you use CA Datacom Datadictionary to modify any attribute of a table that is accessible through SQL, CA Datacom Datadictionary changes the SQL-ACCESS attribute-value to N. You must run the VERIFY or CATALOG function on the table to make CA Datacom Datadictionary change the value back to Y and allow SQL access.

SQL Tables and Logging

Proper execution of SQL statements that cause data change requires all affected tables to have logging enabled. Logging should not be turned off (LOGGING=NO) for any SQL maintenance. This applies to both the CA Datacom Datadictionary definition and the DBUTLTY CXXMAINT option.

Creating SQL Tables

To create SQL tables, you can embed CREATE TABLE statements (one for each table) in an application, or you can submit CREATE TABLE statements through the CA Datacom Datadictionary Interactive SQL Service Facility. You can also create SQL tables through CA Dataquery (see the CA Dataquery documentation for details). To define an SQL table, you must specify the name of the table and the name, the data type and the length of each column in the table.

You can optionally specify if one or more columns are to have a unique value for each row of the table. You can specify this UNIQUE constraint on individual columns and/or a list of columns whose combined values are unique. Using the UNIQUE constraint creates a KEY entity-occurrence in CA Datacom Datadictionary.

Note: Uniqueness is enforced at the key level, not at the column level, that is to say, UNIQUE forces unique values for the entire key and not for the individual columns making up the key.

When you define your table, you can specify the area where the table data is to reside. If you do not specify the area name, the data is placed in a default area which is specified at installation for your convenience. If you want to store your data in an area other than the default, see your Database Administrator to have a specific area defined for your use.

When the CREATE TABLE statement is executed (either embedded or through the CA Datacom Datadictionary Interactive SQL Service Facility), the table, columns, and any KEY entity-occurrences generated by use of the UNIQUE option are defined to CA Datacom Datadictionary in PRODuction status and cataloged to the CA Datacom/DB Directory (CXX).

Using Existing Tables

Existing tables which were not created by the SQL Manager can be accessed by SQL if they adhere to the following rules:

1. The table definition must exist in the CA Datacom Datadictionary.
2. The value in the table's SQLNAME attribute must be a valid SQL table name. The valid character set includes A-Z, 0-9, $, #, @, and _ (underscore). The area and database in which the table resides must also have valid SQLNAMEs.
3. The value in the table's AUTHID attribute must be a valid authorization ID.
4. Field names must have valid SQL column names specified for the SQLNAME attribute.
5. If the table includes group fields and you are *not* using DATACOM VIEWs, only the lowest-level, simple fields may be accessed using SQL. For information about DATACOM VIEWs, see DATACOM VIEWs (see page 118).
6. Data types other than the following types are treated as character fields by SQL. See SQL Data Type Support for All CA Datacom/DB Tables for more information.
   - C (character)
   - B 2 (small integer, 2-byte binary, signed)
   - B 4 (long integer, 4-byte binary, signed)
   - L (float, signed)
   - D (packed decimal, signed; decimal, unsigned; decimal, positive)
   - N (zoned decimal, signed; numeric, unsigned; numeric, positive)
   - DATE (B 4 (binary, length=4) SEMANTIC-TYPE=SQL-DATE)
   - TIME (B 3 (binary, length=3) SEMANTIC-TYPE=SQL-TIME)
   - TIMESTAMP (B 10 (binary length=10) SEMANTIC-TYPE=SQL-STMP)

   Note: DATE, TIME, and TIMESTAMP are stored as binary data but are automatically converted to character strings in SQL. See Character String Literals for more information. See the *CA Datacom/DB Database and System Administration Guide* for an explanation of how DATE, TIME, and TIMESTAMP data types are stored in CA Datacom/DB. If you are accessing an SQL DATE, TIME, or TIMESTAMP with a non-SQL command, you must perform the conversion from the internal format yourself.

See Data Types for more information about SQL data types.

7. If the repeat factor of a group field is greater than one (for example, let the repeat factor be represented by the letter R), the entire field (all R elements) is treated as one-character column by SQL. Arrays on group fields are not supported. If the table includes arrays and you are not using DATACOM VIEWs, the entire array is treated as one-character field by SQL.

   Note: To process this as group field using a DATACOM VIEW (see DATACOM VIEWs (see page 118)), you would need to create a "redefine" that either defines each subfield R times or redefines the group field as a CHAR column with a repeat factor of R. In the latter case, the SQL SUBSTR (substring) and CAST WITHOUT CONVERSION functions can be used to extract the desired sub-fields. Parallel arrays of simple repeating fields are supported.

8. No variable-length fields can exist in the table definition.
9. Redefined fields can exist in the table definition, but SQL ignores the REDEFINES attribute.
10. The value of the table's SQL-INTENT attribute must be set to Y to mark that the table is to be accessed by SQL. The table must successfully pass the verification for SQL access during the catalog operation before it can be accessed by SQL.
11. After you complete any modifications to make the tables SQL accessible, the table must be copied to PROD status and cataloged to the CA Datacom/DB Directory (CXX).

The SQL CREATE TABLE statement cannot create a remote, partitioned or replicated table. For remote tables, SQL access requires a complete duplicate definition of a remote table in CA Datacom Datadictionary, and version control enforcement is not available (version control enforcement helps ensure that remote definitions are synchronized with the local active definition).

Populating SQL Tables

Before you can access data in a newly defined SQL table, you must populate the table. You can use any traditional method to load the data, or you can use the SQL INSERT statement.

Accessing SQL Tables

Once your table is populated, you can access data using DML statements you:

- Embed in applications (see Embedding SQL Statements in Host Programs)
- Submit through CA Datacom Datadictionary Interactive SQL Service Facility (see Using the Interactive SQL Service Facility)
- Submit through CA Dataquery (see the *CA Dataquery User Guide*)

Note: You cannot use SQL Data Definition Language (DDL) statements to modify tables defined in CA Datacom Datadictionary as SQL read-only. For more information on SQL read-only tables, see SQL Read-Only.

Selecting and Manipulating Data

You can select the data you want to work with using the SQL SELECT statement. Since SQL is a powerful language, the SELECT statement can accommodate complex constructs. However, if you are a new SQL user, you may want to start with simple language constructs, especially using the SELECT and specifying search conditions.

The most common data manipulation operations involve inserting, updating and deleting. Specifying the data to be manipulated can involve cursor and non-cursor operations.

In application programs, any changes to data can be committed by issuing the COMMIT WORK statement. Changes which have not been committed can be backed out by issuing the ROLLBACK WORK statement. CA Datacom Datadictionary and CA Dataquery automatically handle transaction commits and rollbacks.

Specifying Preprocessor Options

Each application with embedded SQL statements must have a CA Datacom/DB access plan. The plan contains information required by CA Datacom/DB about your program and information about each SQL statement you have embedded.

The plan is built when you submit your application to the CA Datacom/DB SQL Preprocessor. The Preprocessor has options which you can optionally specify or let default to determine how the Preprocessor processes the SQL statements and to control certain aspects of the application's execution.

The Preprocessor options allow you to specify criteria for your application, such as if the SQL statements must include only ANSI standard constructs, or if CA Datacom/DB extensions to SQL are allowed (extended mode). You can also name the plan for your application, specify the plan's authorization ID, designate the isolation level for your application, indicate when the plan is to close, or specify an I/O limit interrupt value for SQL statements.

Preparing Programs

If you are embedding SQL statements in an application program, you must distinguish SQL statements from source code and place the SQL statements in the appropriate division or section of the source program.

Your application can contain an INCLUDE directive to include a member from an include library, if you have coded the Preprocessor option to allow CA Datacom/DB extensions to SQL.

In an application with embedded SQL, you can indicate the action to take when an exception condition is encountered by including WHENEVER statements.

After you have coded your program, you must submit it to the CA Datacom/DB SQL Preprocessor. You must compile and link edit your program along with an SQL User Requirements Table and the host variable processor, DBXHVPR.

Mixed Mode Programming

Mixed mode programming is the embedding of SQL statements in application programs where native CA Datacom/DB record-at-a-time and/or set-at-a-time commands are also coded.

The embedded SQL statements can be either CA Datacom/DB SQL statements or IBM DB2 SQL statements *but not both,* that is to say, CA Datacom/DB SQL and DB2 SQL calls cannot be made from the same program. All of the SQL statements in a program must be processed either by IBM DB2 or CA Datacom/DB but not both because both require the source program to embed the SQL statements in the same special statements (EXEC-SQL and END-SQL), and both require the source to be manipulated by their own Preprocessor. If the source was processed through a CA Datacom/DB Preprocessor first, for example, it could therefore not later be passed through an IBM Preprocessor, because at that point there would be no special statements left to process.

To make native CA Datacom/DB calls and CA Datacom/DB SQL calls from the same application program, see Embedding SQL Statements in Host Programs.

The following requirements must be met to make native CA Datacom/DB calls and IBM DB2 SQL calls from the same application program:

1. The DBURINF User Requirements Table macro must have OPEN=USER and USRNTRY=program-id.
2. The COBOL program must be compiled with the compiler option NODYNAM.
3. The link-edit step must have:

```
INCLUDE SYSLIB(-urt-name)
ENTRY program-id
NAME program-id(R)
```

Statement Execution Table

The following table summarizes the methods by which each SQL statement can be executed. An asterisk indicates that the statement is executable by that method.

SQL Statement	**CA Datacom Datadictionary Interactive SQL Service Facility *(Interactive)***	**In an application program prepared using an CA Datacom/DB SQL Preprocessor *(Embedded)***	**CA Dataquery *(SQL & Batch Modes)***
ALTER TABLE	*	*	*

SQL Statement	CA Datacom Datadictionary Interactive SQL Service Facility *(Interactive)*	In an application program prepared using an CA Datacom/DB SQL Preprocessor *(Embedded)*	CA Dataquery *(SQL & Batch Modes)*
CALL	YES (if no parms are passed)	*	YES (if no parms are passed)
CLOSE		*	
COMMENT ON	*	*	*
COMMIT WORK	*	*	
CREATE INDEX	*	*	*
CREATE PROCEDURE	*	*	*
CREATE RULE	*	*	*
CREATE SCHEMA	*	*	
CREATE SYNONYM	*	*	*
CREATE TABLE	*	*	*
CREATE TRIGGER	*	*	*
CREATE VIEW	*	*	*
DECLARE CURSOR		*	
DECLARE STATEMENT	*		
DELETE (positioned)		*	
DELETE (searched)	*	*	*
DESCRIBE		*	
DROP INDEX	*	*	
DROP RULE	*	*	*
DROP PROCEDURE	*	*	*
DROP SYNONYM	*	*	*
DROP TABLE	*	*	*
DROP TRIGGER	*	*	*
DROP VIEW	*	*	*
EXECUTE		*	

SQL Statement	CA Datacom Datadictionary Interactive SQL Service Facility *(Interactive)*	In an application program prepared using an CA Datacom/DB SQL Preprocessor *(Embedded)*	CA Dataquery *(SQL & Batch Modes)*
EXECUTE IMMEDIATE		*	
EXECUTE PROCEDURE	YES (if no parms are passed)	*	YES (if no parms are passed)
FETCH		*	
GRANT	*	*	*
INSERT	*	*	*
LOCK TABLE	*	*	
OPEN		*	
PREPARE		*	
REVOKE	*	*	*
ROLLBACK WORK	*	*	
select-into statement		*	
select-statement	*	(use DECLARE CURSOR)	*
full-select statement	(part of the select-statement)	(part of the select-statement)	(part of the select-statement)
subselect	(part of full-select statement)	(part of full-select statement)	(part of full-select statement)
SET CURRENT SQLID		*	
UPDATE (positioned)		*	
UPDATE (searched)	*	*	*
WHENEVER		*	

Dynamic SQL

When *static* SQL cannot satisfy the functional requirements of a program, you can use *dynamic* SQL for the statements listed in the table in the section PREPARE.

Static SQL

In static SQL, you embed SQL statements (the SQL source) in a host language program and bind them before executing the program. This makes the statements *static*, that is, when you write the SQL source into the program, the format of the SQL statements are known to you and do not change when the program is executed. If you therefore know, when you write the program, the type of SQL statements (such as SELECT, UPDATE, INSERT) to be used and the table names and column names of the data to be accessed, static SQL efficiently provides what you need. But if your program requires complete flexibility, that is, if it needs to execute so many different types and structures of SQL statements that it is impossible for it to contain a model of each one, use dynamic SQL.

Do not think of static SQL as being totally unflexible. If your static SQL statements include host variables that your program changes, you can use static SQL and still enjoy a reasonable amount of flexibility. Consider the following COBOL example. Notice that the values of WRKRID and NEWPRAT are reset each time the UPDATE statement is re-executed, so that it updates the performance ratings of as many workers as required with whatever values are needed.

```
MOVE PRFRAT-7 TO NEWPRAT.
MOVE 000090 TO WRKRID.
EXEC SQL
  UPDATE DRAKE06.WRK
    SET RATING = :NEWPRAT
    WHERE WRKNO = :WRKRID
END-EXEC.
```

Dynamic SQL

In dynamic SQL, you do not write SQL source statements into the application program. Instead, you use variables in the host language to contain the SQL source. The SQL statements are then *dynamically* prepared and executed within the program as it runs and can change one or more times during the program's execution. This means you do not need to have complete knowledge of a dynamic SQL statement's full format at the time you write the program.

For example, if your program needs to allow for a large variety of selection criteria, with static SQL a small set of criteria can be used to select a table's rows while the rest of the criteria are compared against the return rows, but with dynamic SQL an SQL WHERE clause can be generated to drive the program and match the criteria exactly. (A WHERE clause produces an intermediate result table by applying a search condition to each row of a table R, the result of a FROM clause. The result table contains the rows of R for which the search condition is true. See the description in Subselect.

Dynamic SQL in CA Datacom/DB

Dynamic SQL in CA Datacom/DB is compatible with IBM DB2's dynamic SQL. Application programs that have been using DB2's dynamic SQL can precompile successfully and execute equivalently under the DB2 mode of CA Datacom/DB.

Five statements in CA Datacom/DB support dynamic SQL:

- DECLARE STATEMENT (see DECLARE STATEMENT)

  For syntax compatibility with other SQL implementations, the DECLARE STATEMENT is accepted by the CA Datacom/DB Preprocessor for SQL, but CA Datacom/DB ignores everything after the keyword STATEMENT up to the end-of-statement delimiter. CA Datacom/DB functionality is not affected.

- DESCRIBE (see DESCRIBE)

  This statement obtains information about a specified table or view, or about a statement that has been prepared for execution by the PREPARE statement.

- EXECUTE (see EXECUTE)

  This statement executes an SQL statement that has previously been prepared for execution by the PREPARE statement.

- EXECUTE IMMEDIATE (see EXECUTE IMMEDIATE)

  This statement prepares an executable form of an SQL statement from a character string form, executes the SQL statement, and then destroys the executable form.

- PREPARE (see PREPARE)

  This statement creates an executable SQL statement from a character string form of the statement. The executable form is called a prepared statement.

Three other CA Datacom/DB SQL statements also support dynamic SQL:

- DECLARE CURSOR (see DECLARE CURSOR)
- FETCH (see FETCH)
- OPEN (see OPEN)

INCLUDE Directive

For users of PL/I and Assembler, the INCLUDE directive attaches special meaning to a member name of SQLDA, referring to the SQL Descriptor Area (for more information on the SQLDA, see SQL Descriptor Area (SQLDA)). When INCLUDE SQLDA is specified, the Preprocessor for PL/I or Assembler includes the description of an SQL Descriptor Area (SQLDA) for use by dynamic SQL statements. For more information on the INCLUDE directive, for PL/I see Rules for SQL INCLUDEs in PL/I, or for Assembler see Rules for SQL INCLUDEs in Assembler. For an example SQLDA in COBOL, see Example.

Name Types

Two CA Datacom/DB name types (descriptor-name and statement-name) support dynamic SQL. See Naming Conventions for more information.

Reserved Words

Four reserved words pertain to dynamic SQL, as shown on Reserved Words (see page 29): DESCRIPTOR, EXECUTE, IMMEDIATE, USING.

Parameter Markers

A parameter marker is a question mark (?) that is used in place of a host variable in dynamic SQL statements. The rules for using parameter markers in a prepared statement are given on Rules for Parameter Markers. If a prepared statement contains parameter markers, you must use the USING clause of the EXECUTE statement. For information on the USING clause and about parameter marker replacement, see the PREPARE.

Security Implications of Dynamic SQL

Security checking of a dynamically prepared statement is always done when that statement is executed. When a statement is dynamically prepared, no security checking is done, because no special privileges are required to dynamically prepare SQL statements. But when the dynamically prepared statement is executed, the accessor ID of the executor is checked for the privileges required to do the requested operations on the database.

Using Dynamic SQL in Application Programs

An SQL statement in character string form is accepted as input by (or is generated from) an application program that uses dynamic SQL. To simplify a program that uses dynamic SQL, code it so that it either does not use SELECT statements or only uses SELECT statements that return a known number of values of known types.

When you are coding a program that uses dynamic SQL, but you do not know which SQL statements are to be executed, consider having the program take the following steps:

1. For input data (including parameter markers), translate it into an SQL statement.
2. For the SQL statement,
    a. Prepare it for execution, and
    b. Obtain its description.

3. For SELECT statements, acquire enough main storage to contain the data that is retrieved.
4. Then, either:
    a. Execute the statement, or
    b. Fetch the rows of data.
5. Next, process the information that is returned.
6. And then deal with SQL return codes.

Performance Considerations

Be aware of the following performance considerations with regard to dynamic SQL. When you use dynamic SQL statements, the "runtime overhead" is greater than for static SQL statements, because the processing of the statement is similar to a program that must be preprocessed before it is executed. You may therefore want to limit your use of dynamic SQL to those situations in which its flexibility is required.

Performance is not greatly affected if you only use a few dynamic SQL statements in an application program that takes a long time to execute. The same number of dynamic SQL statements in a program of short duration, however, can affect performance significantly.

The following sections show how the various statement-types (and new variants of statement-types) that make up Dynamic SQL are used to provide an application with the ability to execute SQL statements when those statements are not completely known before the program is executed.

Classes of Use

There are four classes of use for dynamic SQL:

- When no SELECT statements are issued dynamically, dynamic allocation of main storage is not needed. This is the simplest way to use dynamic SQL (see the example on Dynamic SQL for Non-SELECT Statements (see page 46)).
- Use fixed-list SELECT statements when you know ahead of program execution time what kinds of host variables need to be declared to store results. That is, when you have rows that contain a known number of values of a known type, use fixed-list SELECT statements to return them (see the example on Dynamic SQL for Fixed-List SELECT Statements (see page 47)).
- Use varying-list SELECT statements when you do not know ahead of program execution time what kinds of host variables need to be declared to store results. That is, when you have rows that contain an unknown number of values of unknown type, use varying-list SELECT statements to return them (see the example on Dynamic SQL for Varying-List SELECT Statements (see page 48)).
- If you need to have several kinds of dynamic SQL statements (including varying-list SELECT statements, in any of which a variable number of parameter markers might be contained) executed in a program, the program could be said to execute "arbitrary" SQL statements. An example begins on Dynamic SQL for Arbitrary Statement-Types (see page 49).

Note: In addition to the examples in the following sections, see the sample dynamic SQL program on the CA Datacom/DB eSupport website.

Dynamic SQL for Non-SELECT Statements

The simplest use of dynamic SQL is when only non-SELECT statements are to be dynamically executed and the SQLDA does not have to be explicitly used.

Note: Because you know when you code this type of program how many parameter markers are to be included in the statement, you can code the USING clause of the EXECUTE statement with a list of variable names.

The steps taken by a program where only non-SELECT statements are dynamically executed are as follows:

1. Read a statement containing parameter markers (for example, DELETE FROM CUSTOMERS WHERE CUSNO=?) into USERSTR.
2. Do a PREPARE of USERSTMT.

```
EXEC SQL
  PREPARE USERSTMT FROM :USERSTR
END-EXEC
```

3. Read a value for CUSNO from some list.

```
DO UNTIL (CUSNO = 0)
  EXEC SQL
    EXECUTE USERSTMT USING :CUSNO
  END-EXEC
ENDDO
```

4. Read the next value for CUSNO from the list, and so on.
5. Deal with any SQL return codes that indicate errors.

Dynamic SQL for Fixed-List SELECT Statements

In the following example, assume that you know the number and data types of the columns in the SELECT's result table when you code the application program.

To dynamically execute a fixed-list SELECT statement your program must:

1. Load input SQL statement into a data area (as in non-SELECT statement example in Dynamic SQL for Non-SELECT Statements (see page 46)).
2. DECLARE a cursor-name for the statement name.

```
EXEC SQL DECLARE CURSOR1 CURSOR FOR STMT;
```

3. Read or construct an SQL select-statement (of the form SELECT NAME, ZIP FROM CUSTOMERS WHERE...) into host variable USERSTR, then use a PREPARE statement and an OPEN statement as shown.

```
EXEC SQL PREPARE STMT FROM :USERSTR;

EXEC SQL OPEN CURSOR1;
```

   Alternately, if there were always two parameter markers in the statement:

```
EXEC SQL OPEN CURSOR1 USING :PARA1, :PARA2;
```

   Or, to be more flexible, the input host variables could be described by an SQLDA, as in:

```
EXEC SQL OPEN CURSOR1 USING DESCRIPTOR :SQLDA-PARAS;
```

   The application program in this case is required to ensure that the number of host variables described in the SQLDA matches the number of parameter markers in the SQL statement.

4. FETCH rows from result table.

```
EXEC SQL FETCH CURSOR1 INTO :NAME, :PHONE;
```

5. CLOSE the cursor.

```
EXEC SQL CLOSE CURSOR1;
```

6. Deal with any SQL return codes that indicate errors.

Dynamic SQL for Varying-List SELECT Statements

The most complex way to use dynamic SQL for SELECT statements is when you do not know (when you write the application program) the number and data types of the columns in the SELECT's result table. This requires the use of varying-list SELECT statements.

As the example below shows, a program that uses varying-list SELECT statements must do the following:

1. Load input SQL statement into a data area (as in non-SELECT statement example already given).
2. DECLARE a cursor-name for the statement name.

```
EXEC SQL DECLARE CURSOR1 CURSOR FOR STMT;
```

3. Declare a host variable called SMLSQLDA of data type SQLDA, with 100 SQLVARs, SQLN=100 (for example, :SQLSTRING = SELECT COL1 FROM TAB1). If the statement in :SQLSTRING contains parameter markers, allocate and initialize an SQLDA called, for example, SQLDAPARA, which describes the host variables that correspond to the parameter markers. Allocate the storage for these host variables, if necessary.
4. Prepare the variable statement.

```
PREPARE USERSTMT INTO SMLSQLDA FROM :SQLSTRING;

IF SMLSQLDA.SQLD = 0 THEN
    the statement was not a select-statement--ERROR
ENDIF
```

5. You must next determine if you have to allocate a larger SQLDA. The PREPARE caused SMLSQLDA.SQLD to be set to the number of columns in the result table, and SMLSQLDA.SQLDABC is equal to the size in bytes required for an SQLDA big enough to describe that many columns (16 + SQLD * 44). If SMLSQLDA.SQLD is greater than SMLSQLDA.SQLN, the number of columns in the result table is larger than the size allowed for in SMLSQLDA. In this case, no information has been put into the SQLVARs of SMLSQLDA. The SQLD field has been set to the number of columns in the result table, so that an SQLDA of the required size may be allocated.

   The application program should now allocate an SQLDA of the size indicated by SMLSQLDA.SQLD. If we call this full-size SQLDA LRGSQLDA, to get the description of the result table filled in, the application program should then execute a DESCRIBE statement, using LRGSQLDA.

   ```
   EXEC SQL DESCRIBE STMT INTO LRGSQLDA;
   ```

   Now we have an SQLDA that describes (in its SQLVAR section) all of the columns of the result table of the select-statement in SQLSTRING. Examine the SQLDA to allocate storage for a result row of the select-statement. Set the addresses in the SQLVAR entries to point to memory allocated for each entry.

   ```
   EXEC SQL OPEN CURSOR1;
   ```

   Or, if there were parameter markers in the select-statement:

   ```
   EXEC SQL OPEN CURSOR1 USING DESCRIPTOR SQLDAPARA;
   ```

6. FETCH rows from result table and CLOSE the cursor. The clause USING DESCRIPTOR LRGSQLDA names an SQLDA in which the occurrences of SQLVAR point to other areas that receive the values returned by the FETCH. The USING clause can be used here because LRGSQLDA was set up previously (in this example).

   ```
   EXEC SQL FETCH CURSOR1 USING DESCRIPTOR LRGSQLDA;
   ```

7. Close the cursor.

   ```
   EXEC SQL CLOSE CURSOR1;
   ```

8. Deal with any SQL return codes that indicate errors.

Dynamic SQL for Arbitrary Statement-Types

The most complex use of dynamic SQL is when you need to execute, in a program, several kinds of dynamic SQL statements, including varying-list SELECT statements (in any of which a variable number of parameter markers might be contained). Such a program could be said to execute "arbitrary" SQL statements. For example, this kind of program could present a list of choices, such as choices about:

- Operations (select, delete, update)
- Table names
- Columns to select or update

The program could also allow the entering of lists (such as worker ID numbers) by which to control the application of operations.

When you know the number and types of parameters, but you do not know in advance the number of parameter markers (and perhaps the kinds of parameter they stand for):

- Name a list of host variables in the EXECUTE statement *if the SQL statement is not SELECT.*
- Name a list of host variables in the OPEN statement *if the SQL statement is SELECT.*

In either case, the number and types of host variables named by your program must agree with the number of parameter markers (in USERSTMT) and the types of parameter for which they stand. The first variable you use must have the expected type for the first parameter marker in the statement, and so on for the other variables and parameter marker. You must therefore use at least as many variables as the number of parameter markers.

Use a SQL Descriptor Area (SQLDA) when you do not know the number and types of parameters. You can have as many SQLDAs included in your program as you want (there is no upper limit), and they do not all have to be used for the same thing. You can also set up an SQLDA to describe a *set* of parameters.

For purposes of this example, the SQLDA describing a set of parameters is called SQLDAPARA. Its structure is the same as the structure of other SQLDAs, and as in other SQLDA structures, the number of SQLVAR occurrences can vary. But in SQLDAPARA, each occurrence of SQLVAR is used to describe one host variable that replaces one parameter marker at execution time, either when (for a SELECT statement) a cursor is opened, or when a non-SELECT statement is executed. With SQLDAPARA, there must therefore be one SQLVAR for every parameter marker.

In SQLDAPARA you can ignore some of the SQLDA fields, but in the case of other fields, you must fill them before you use an EXECUTE or OPEN statement. See SQLDA (EXECUTE, FETCH, or OPEN Statement).

Following is an example of a program that executes arbitrary SQL statements.

1. To allow for the case in which USERSTMT gets prepared as a select-statement, do this DECLARE statement:

   Note: Alternately, this DECLARE statement could be located (as noted later in this example) in the part of the program labeled: "* The statement is a select-statement. *"

```
PROC HANDLEALL

EXEC SQL DECLARE CURSOR1 CURSOR FOR USERSTMT;
```

2. Allocate a host variable called SMLSQLDA of data type SQLDA, with 100 SQLVARs, SQLN=100.

   Note: 100 in the previous is an arbitrary number. To save static storage it could be much less than 100, but the lower number would result in a lengthened execution time because of the increased likelihood of needing the second DESCRIBE that is shown later in this example.

```
SQLSTRING := the SQL statement, constructed in some way

IF the statement in SQLSTRING has parameter markers THEN

    Analyze the SQL statement: find the parameter markers
      and decide what host variables could be used to
      contain values for each one.  Allocate the host
      variables and indicator variables if necessary.
      (*These parameter markers all describe input host
      variables, values that must be passed to the DBMS when the
      statement is executed.*)

   Declare a host variable called SQLDAPARA of data type
      SQLDA and fill it in to correctly describe and
      point to host variables that correspond to the
      parameter markers.

ENDIF

EXEC SQL PREPARE USERSTMT INTO SMLSQLDA FROM :SQLSTRING;
```

   Or alternately:

```
EXEC SQL PREPARE USERSTMT FROM :SQLSTRING;

EXEC SQL DESCRIBE USERSTMT INTO SMLSQLDA;
```

3. The output result of the PREPARE INTO statement (or the PREPARE and DESCRIBE statements) is that the SMLSQLDA is filled in by the DBMS to describe the result table of the statement in USERSTMT. If the statement in USERSTMT is not a select-statement, there is no result table.

```
 IF SMLSQLDA.SQLD = 0 THEN

     (* The statement is not a select-statement. *)

     IF the statement in SQLSTRING has no parameter markers THEN

         EXEC SQL EXECUTE USERSTMT;

     ELSE

         IF the statement in SQLSTRING does have parameter
          markers THEN

             EXEC SQL EXECUTE USERSTMT USING DESCRIPTOR SQLDAPARA;

         ENDIF

     ENDIF

 ELSE

     (* The statement is a select-statement. *)

     (*NOTE: The DECLARE CURSOR1 CURSOR FOR USERSTMT statement could
       go here, if desired.  It is not executed, but must
       appear physically in the application program source
       before any other statement uses the cursor name.*)

     (*If needed, allocate a bigger SQLDA:*)

     IF SMLSQLDA.SQLD > SMLSQLDA.SQLN THEN
```

The number of columns in the result table is larger than the size allowed for in SMLSQLDA. In this case, no information has been put into the SQLVARs of SMLSQLDA. The SQLD field has been set to the number of columns in the result table, so that an SQLDA of the required size may be allocated. The application program must now allocate an SQLDA of the size indicated by SMLSQLDA.SQLD. In this example, this full-sized SQLDA is called LRGSQLDA.

4. Allocate a host variable of data type SQLDA, called LRGSQLDA:

```
LRGSQLDA, SQLN := SMLSQLDA.SQLD, SQLDABC :=
16 + SQLN * 44, with SQLN SQLVAR's.
```

5. To get the description of the result table filled in, the application program must now execute a DESCRIBE statement using LRGSQLDA.

```
EXEC SQL DESCRIBE USERSTMT INTO LRGSQLDA;

ENDIF
```

We now have an SQLDA that describes, in its SQLVAR section, all of the columns of the result table of the select-statement in SQLSTRING.

6. Allocate storage for a result row of the select-statement by examining the SQLDA (either SMLSQLDA or LRGSQLDA). Set the addresses in the SQLVAR entries to point to the host variables allocated for each column of the result table.

```
IF the statement in SQLSTRING has no parameter markers THEN

    EXEC SQL OPEN CURSOR1;

ELSE

    EXEC SQL OPEN CURSOR1 USING DESCRIPTOR SQLDAPARA;

ENDIF

DO WHILE SQLCODE NOT = 100

    IF LRGSQLDA was allocated

        EXEC SQL FETCH CURSOR1 USING DESCRIPTOR LRGSQLDA;

    ELSE

        EXEC SQL FETCH CURSOR1 USING DESCRIPTOR SMLSQLDA;

    ENDIF
```

7. At this point, the program can decide whether to delete the row (that was just read) or to update it. If the program wants to update or delete the current row, it can build the UPDATE or DELETE statement and execute it dynamically, as follows:

```
          IF an update is desired THEN

              VAR5 := the UPDATE WHERE CURRENT statement,
                 constructed in some way, for example,
                      'UPDATE TABLEX SET COL1 = 5
                       WHERE CURRENT OF CURSOR1'

              EXEC SQL PREPARE S2 FROM :VAR5 END-EXEC;

              EXEC SQL  EXECUTE S2  END-EXEC;

          ELSEIF a delete is desired THEN

              VAR5 := the DELETE WHERE CURRENT statement,
                 constructed in some way, for example,
                      'DELETE FROM TABLEX
                       WHERE CURRENT OF CURSOR1'

              EXEC SQL PREPARE S2 FROM :VAR5 END-EXEC;

              EXEC SQL EXECUTE S2 END-EXEC;

          ENDIF

      ENDDO

      EXEC SQL CLOSE CURSOR1;

   ENDIF

   ENDPROC HANDLEALL
```

8. Deal with any SQL return codes that indicate errors.

Other Tasks

After your SQL tables are defined to the CA Datacom Datadictionary, you can create views based on one or more tables. You can also create a view based on one or more views. Views allow you to retrieve only that data which is significant for your purposes.

You can create synonyms for your tables and views, or for tables and views owned by other authorization IDs. Synonyms are short names for tables or views.

You can define a new index on one or more columns of a base table.

When making changes to data, you can control access to SQL tables through the isolation level Preprocessor option, or with the LOCK TABLE statement.

If you no longer need a table, view, synonym, or index, you can drop the SQL object using the DROP statement. If you have created an SQL object simply for testing purposes or only for the run of an application, you can use the DROP statement to remove this object from CA Datacom Datadictionary.

Important! If you remove an SQL object from CA Datacom Datadictionary, you must recreate the object you dropped if you want to use it again.

If a table is dropped, all views and synonyms based on that table are removed from CA Datacom Datadictionary. The table definition is removed from the CA Datacom Datadictionary and the CA Datacom/DB Directory (CXX), the table data is deleted and the space is reclaimed.

If a view is dropped, all views and synonyms based on the view are removed from CA Datacom Datadictionary.

If a synonym is dropped, only that synonym is removed from CA Datacom Datadictionary.

If you drop an index, all plans dependent on the indexed table are marked invalid. You can run a CA Datacom Datadictionary Relationship Report to find out what plans are dependent on a table. See the *CA Datacom Datadictionary Batch Reference Guide* for information about running a Relationship Report.

SQL Status Tables

The following tables provide information about the current status of the SQL subsystem:

- SQL_STATUS (SQS)
- SQL_STATUS_CURRENT (SQC)
- SQL_STATUS_PLAN (SQP)
- SQL_STATUS_URT (SQU)
- SQL_MISC_STATS (SQM)
- SQL_SQLCODES (SQQ)

These SQL status tables are located in the dynamic system tables database. See the *CA Datacom/DB System Tables Reference Guide* for detailed information.

Procedures and Triggers

Procedures and triggers provide CA Datacom with substantial flexibility in building "thin client/fat server" applications, enforcing business rules, implementing additional security functionality, and even providing functionality to enable a relational view of nonrelational data items.

Overview

External Procedures are user-written programs (written using LE-conforming Assembler, COBOL, PL/I, or C) which execute inside the Multi-User Facility as a separate subtask. They can be coded to perform almost any task and generally contain SQL statements. SQL procedures consist of user-written program logic composed of SQL statements and contained entirely within the CREATE PROCEDURE statement. Both types can be executed explicitly using CALL or EXECUTE PROCEDURE statements and can be triggered implicitly by user-specified database management system events such as INSERT, UPDATE, and DELETE. For information specific to each procedure type, see CREATE PROCEDURE.

Triggers depend completely on the SQL procedure to perform the various "triggered" activities. Triggers themselves have a relatively limited logic implementation. A trigger is set to be "fired" when an insert, delete, or update of a row in the selected table occurs. The trigger is fired regardless of where (CICS, batch, Server, and so on) or how (record-at-a-time, set-at-a-time, SQL) the maintenance command was issued. The trigger is not fired when a read (or read for update command) is processed.

Each trigger can be tailored to fire only when specific data values exist. The trigger can also select whether it should fire before or after the event occurs. Selecting "before" gives you the chance to review the process before the maintenance has occurred, while "after" allows you to Trigger the event after the maintenance has occurred.

LUW Control

The trigger and its procedure operate under the same logical-unit-of-work (LUW) as the task that caused the trigger to fire. Any failure in execution of the trigger and its procedure causes the maintenance command that fired the procedure to receive a return code and the maintenance command is rejected by the database. In addition, any database work done by the triggered procedure (inserts, deletes, updates) is also done under the same LUW as the application that fired the trigger. Any subsequent commit issued by the application that caused the trigger to fire also commits any database maintenance done by the fired procedure. Likewise, a subsequent ROLLBACK issued by the application also rolls back any triggered changes.

Note: Triggers and their associated procedures cannot contain any commit or rollback logic.

Thin Client/Fat Server

You can write External Procedures in LE-conforming Assembler, C, PL/I, or COBOL as standard executing programs. These programs accept a list of input variables (SQL columns), perform any number of program and/or database functions, then return a list of output variables (SQL columns). This functionality allows you to create procedures that can combine a wide variety of program functions into one SQL call.

For example: A client/server application currently processes a customer order by issuing multiple SQL statements to the database region to:

- Verify the customer number
- Obtain the customer's credit limit/availability
- Validate that the ordered item(s) is in stock
- Validate that the customer has enough credit to order the items
- Enter the order in the system

With a External Procedure in SQL, the client/server application can issue a single SQL call to the External Procedure with the appropriate information (Customer number, Item number(s), Quantity). The database server (Multi-User Facility) loads the appropriate LE-conforming program as a subtask in the Multi-User Facility address space, passes over the input variables, and waits to receive the return information. On completion, the LE-conforming program hands back the output variables to the Multi-User Facility, then the Multi-User Facility returns them to the calling program and terminates the Multi-User Facility subtask. If the program fails, the subtask in the Multi-User Facility address space goes away, and the client SQL call receives an SQL return code. If the program completes but does not provide the expected number of return variables, a different SQL code is returned.

Enforcing Business Rules

Since the procedure is a standard LE-conforming program, processing logic can be incorporated in the procedure to enforce a wide variety of business rules. In the previous example, the procedure validated that the customer had appropriate credit before allowing the order to be placed. This is a simple example of business rule enforcement. More complex rules or even complex data calculations can be included in the procedure code to insure that functions such as "Calculation of product cost" or "Standard deviation" are applied consistently within an environment.

When combined with SQL triggers, you can use procedures to enforce business rules "triggered" by a database event, such as the addition of a row to a table or the update of a column within a selected table.

For example: Whenever a new customer is added to the system, the addition triggers the procedure ORDER_CREDIT_REPORT and also triggers the procedure VALIDATE_TEMP_CREDIT_LIMIT. This allows your business to make sure that credit reports are ordered on a timely basis and that a temporary credit limit is established.

Enhanced Security

Along the same lines, you can use triggers and procedures to do validation any time selected data rows or fields are updated. For example, anytime the PAYROLL_RATE field is updated, it could trigger the procedure CHECK_PAYROLL_RATE_CHANGE.

Automatically Generating Alerts

You can use triggers with procedures to generate alerts when "out-of-bounds" business conditions exist. One example is to generate a re-stock order for the warehouse when an inventory items' shelf quantity drops below a minimum shelf amount. A more complex job is to calculate the minimum amount of stock necessary and initiate a product reorder process to ensure that product does not go out of stock before it is replenished (by a new shipment). Since the procedure is actually an application program, subroutines could be initiated to provide physical alerts such as a console message, fax message, and so on.

Relational View of Nonrelational Data

Many open systems products used by clients today do not provide detail level data manipulation capabilities. In some cases, this can limit the products' ability to see "nonrelational" data as relational columns.

For example, a CA Datacom/DB table has a column defined as:

```
02 MONTH-SALES OCCURS 12 TIMES PIC S9(6)V99
```

An SQL view of this table includes the field MONTH_SALES as a CHAR(96) column. If the product viewing this column cannot redefine the data item, this data becomes unusable to the end user. If this data is important, a External Procedure could be written (in COBOL) to use SQL to retrieve the CHAR(96) column and move it to working storage, where the program language could be used to extract the 12 distinct values and place them in 12 columns SQL can use. These columns with other data items would be returned as part of the output variable list.

Summary

Procedures and triggers can provide a powerful tool for data and database administration and access. However, they should also be used with caution. Replacing a simple call from a CICS transaction with an execute procedure would probably cause a performance degradation. Similarly, using procedures to do simple security checks or to enforce access rules when standard system security products are available may not prove worthwhile. However, triggers and procedures can provide great value when used to enforce complex or highly sensitive business rules.

For detailed information about the SQL statements related to procedures and triggers, see the following:

- CREATE PROCEDURE (see CREATE PROCEDURE)
- CALL/EXECUTE PROCEDURE —without SET processing support (see CALL/EXECUTE PROCEDURE)
- DROP PROCEDURE (see DROP)
- CREATE TRIGGER/RULE—row level only (see CREATE TRIGGER/RULE)
- DROP TRIGGER/RULE (see DROP)

For examples showing the use of procedures and triggers, see Examples: Creating a Procedure (see page 73) and Example: Calling a Procedure (see page 92).

Note: Some parts of CA Datacom SQL statement syntax are extensions to the ANSI SQL3 standard and are therefore rejected by SQL when used under ANSI and FIPS SQLMODEs.

SQL Procedures

For information on the SQL Procedures feature that was added at r11 SP4, see CREATE PROCEDURE.

External Security Support for Procedure/Trigger Creation and Execution

Users executing on externally secured systems cannot create, execute, or drop procedures or triggers if the appropriate access rights have not been granted to them. Plan security secures procedure and trigger execution in CA Datacom/DB Version 10.0. CREATE PROCEDURE, CREATE TRIGGER, DROP PROCEDURE and DROP TRIGGER are secured using the DTADMIN external access.

Trigger Execution for Record-at-a-Time Maintenance

If you are executing record-at-a-time requests to maintain tables for which triggers are defined, be aware that the triggers execute as specified in the SQL CREATE TRIGGER statement. Any access method whose maintenance requests are routed through SQL causes triggers to fire.

Transaction Integrity

Any database maintenance performed by CA Datacom SQL during the execution of a procedure becomes a part of the transaction that caused the procedure to be executed. A procedure cannot issue COMMIT or ROLLBACK statements because that would terminate the transaction under which it was called. Any work performed by a procedure through means other than CA Datacom SQL is not integrated with the transaction state of the transaction that instigated procedure execution.

When a trigger is created, SQL plans using the table involved are marked invalid and are automatically rebound the next time they are read from the DDD table. Plans currently in memory do not recognize the added trigger. In general, a plan is flushed from memory when all applications using it have completed (see the information about the PLNCLOSE= preprocessor option in Description of Options). Navigational (native DB non-SQL) commands recognize new triggers only when the application's OPEN for the table in question occurs after the CREATE TRIGGER has completed.

CA Datacom/DB Utility (DBUTLTY) functions MASSADD and DBTEST, when executed with the MULTUSE=YES keyword, invoke triggers as would any other application. All other DBUTLTY functions ignore any trigger definitions and perform the maintenance as directed. These functions include LOAD, MASSADD (running with MULTUSE=NO), and all forms of RECOVERY.

Triggers are not activated by the restart process performed during the Multi-User Facility startup processing or by transaction backout (ROLLBACK) processing performed to reverse a failed transaction.

Any Single User execution (we do not recommend using Single User execution) ignores triggers.

Subroutine Calls Inside Procedures

Procedures may call subroutines that perform non-CA Datacom related tasks, defined as any task that does not cause any piece of CA Datacom code to execute. Following are the rules for CA Datacom related subroutines:

- A subroutine may not contain calls to CA Datacom SQL.
- A subroutine may not be a procedure. Note, however, that a procedure mainline *is* allowed to call procedures using the CALL PROCEDURE and EXECUTE PROCEDURE statements.
- A subroutine may not contain record-at-a-time, set-at-a-time, or other calls that trigger CA Datacom code to execute.
- Because we do not guarantee that CA Datacom physically prevents a subroutine from making a record-at-a-time, set-at-a-time, CA Datacom SQL, or other illegal call, making such a call is not recommended. Illegal calls not only attempt to execute under a different transaction than that of the caller, but could produce unexpected results, unexpected effects, and abnormal terminations for which we cannot be held responsible. **Therefore, make certain that you do not use any illegal calls.**

Restrictions

Following is a partial list of functions that are not supported. If a feature, API, or other item does not appear on this list, its absence does not imply support for it. CA reserves the right to add items to this list or change or delete them at any time.

- Non-SQL requests (record-at-a-time, set-at-a-time, or other) are forbidden inside procedures.
- Procedures cannot contain COMMIT or ROLLBACK statements.
- Requests that trigger execution of any CA Datacom Transparency product are forbidden inside procedures.
- Procedures cannot issue INSERT, UPDATE, or DELETE statements against any CA Datacom DL1 Transparency-constrained table.
- COBOL, PL/I, and C procedures must be made Language Environment (LE) conforming by being written and compiled using the Language Environment. In z/OS, support for procedures and triggers requires a minimum of z/OS Version 2 Release 5 and compatible Language Environment for z/OS with the following IBM compiler products: z/OS C/C++ (only the C subset is supported), COBOL for z/OS, and PL/I for MVS. Assembler procedures must also be made LE-conforming by use of the CEEENTRY and associated macros. A z/VSE operating system is required, along with IBM Language Environment for z/VSE and compatible compilers.
- The depth of nesting of recursive procedure execution, whether procedures are triggered or called explicitly, is limited. See CALL/EXECUTE PROCEDURE for more information.

Multi-User Facility Considerations for Procedures

This section provides considerations for executing and coding procedures to run in the CA Datacom/DB Multi-User Facility.

Note: Because SQL has no way of verifying the compatibility of the user-written code with the procedure defined by the CREATE PROCEDURE statement, it is the sole responsibility of the creator of the procedure to help ensure that the CREATE PROCEDURE statement precisely reflects the parameter list expected by the user-written program. Failure to properly coordinate parameter lists can cause the procedure subtask to abnormally terminate.

Certain tasks need to be performed to execute procedures. Do the following tasks before bringing up the CA Datacom/DB Multi-User Facility.

- Add to the Multi-User Facility library concatenation the Language Environment (LE) runtime and associated language libraries that are needed to execute the procedures. For z/OS, place these ahead of the CA Common Services for z/OS runtime libraries. For z/VSE, place these ahead of the CA CIS (Common Infrastructure Services) runtime libraries.
- Add to the Multi-User Facility library concatenation the libraries containing the programs to be executed as procedures, with any associated subroutines.
- Add an appropriate PROCEDURE Multi-User startup option to the Multi-User Facility SYSIN. Code procedure nests and subtasks carefully.
- Add the appropriate Language Environment (LE) and associated language support data sets to the Multi-User Facility startup job.
- Modify the Language Environment (LE) parameter style exit routine for COBOL, IGZEPSX, to enable the code to provide the same parameter list processing that was done when running VS COBOL II runtime with the ATTACH SVC on MVS. This allows for passing Register 1 and the parameter list without change to the main COBOL procedure program, instead of having the parameter list style determined by Language Environment.

- Make certain you have upgraded Language Environment for z/OS to at least the V2R5.0 release level.

  Note: Support for procedures and triggers requires a minimum of z/OS Version 2 Release 5 and compatible Language Environment for z/OS with the following IBM compiler products: z/OS C/C++ (only the C subset is supported), COBOL for z/OS, and PL/I for MVS.

When coding, compiling, and link editing programs that are to execute as procedures, adhere to the following guidelines:

- Do not modify in your procedure program the Language Environment (LE) user area fields (not to be confused with the PL/I user area). These are set and queried by the procedure processor and interface in the Multi-User Facility by use of the Language Environment CEE3USR callable service.
- All procedure programs with embedded SQL statements must be LE-conforming and must be link edited with the procedure interface DBXPIPR.
- All procedure programs must be coded and link edited as RENT and NODYNAM (no dynamic calls) with AMODE(31) and RMODE(ANY).
- Ensure that the link-edit step receives a return code of 0. Any return code greater than 0 indicates a possible error that could lead to a Multi-User Facility abend. A possible error could be having included an SQL User Requirements Table to resolve the DBNTRY entry point. For example, do not include DBSBTPR. All CA Datacom/DB entry points should be resolved by the inclusion of DBXPIPR, and duplicates should not occur. The exception is the include for DBXHVPR to resolve COBOL host variables.
- Modify your installation defaults for LE to specify the following recommended settings, or include with each procedure program link-edit a CEEUOPT module with these settings:

```
ABTERMENC=(ABEND)
ALL31=(ON)
ANYHEAP=(1K,1K,ANYWHERE,FREE)
BELOWHEAP=(1K,1K,FREE)
HEAP=(32K,32K,ANYWHERE,FREE,8K,4K)
LIBSTACK=(1K,1K,FREE)
STACK=(4K,4K,ANY,KEEP)
STORAGE=(00,NONE,00,0K)
TERMTHDACT=(UADUMP,,32)
TRAP=(OFF)
```

Note: While these options are recommendations, they may not be appropriate for your site or may need to be tuned according to your procedure execution environment within the Multi-User Facility. Run a typical procedure with the RPTSG=(ON) option in CEEUOPT and tune accordingly.

Number of Procedure TCBs

Running Language Environment (LE) subtasks requires a substantial but *not predictable* amount of resources primarily related to LSQA memory and address space 24-bit memory. Errors are possible when a shortage exists of either kind of memory. A shortage occurring in some locations can cause return codes to be received. Shortages in other locations can cause:

- Abend failures in the subtask, and
- Recursion loops.

It is possible for these conditions to cause the Multi-User Facility to be terminated by the operating system.

The Multi-User Facility cannot prevent all outages relating to user procedures and the Language Environment. Because it is not possible to predetermine requirements sufficiently to prevent all errors from occurring, plan to *stress test* your environment and configuration, running less TCBs than you expect would work.

Performance Considerations

Each request to execute a procedure attaches an operating system subtask to a CA Datacom/DB *stub* module that fetches the LE program. At completion this is deleted and detached. This overhead, when done frequently, can be substantial and needs to be considered when deciding to implement procedures.

We recommend running with no more than 20 TCBs until testing proves that your environment can accommodate more.

Note: For more information, see the section on the PROCEDURE Multi-User startup option in the *CA Datacom/DB Database and System Administration Guide*.

Parameter Styles and Error Handling

The PARAMETER STYLE clause of the CREATE PROCEDURE statement defines how parameters are passed between an application program, or a trigger, and the procedure that is being called. How errors are handled also depends upon the parameter style chosen.

GENERAL

This parameter style specifies that the user parameter list is passed to the procedure devoid of null indicators (nulls are not allowed). Since no formal method is provided for passing error information back to the caller, the success or failure of the CALL procedure statement is determined by the contents of the SQL internal SQLCODE variable following the last SQL request made by the procedure. This also applies to parameter style GENERAL WITH NULLS.

GENERAL WITH NULLS

This parameter style differs from GENERAL only in that a null indicator is passed to the procedure for each user parameter.

DATACOM SQL

This parameter style passes nulls to the procedure as does GENERAL WITH NULLS and SQL, but it also passes some additional parameters. These parameters are modeled after those passed for the ANSI SQL3 parameter style SQL, but with this difference: instead of a SQLSTATE, DATACOM SQL passes an SQLCODE in the corresponding parameter.

Following are the additional parameters for parameter style DATACOM SQL (the first four are modeled after SQL3):

- SQLCODE—passed to the procedure as 0 and used to set the SQLCODE of the CALL PROCEDURE statement on output.
- A variable-length character string containing the name of the procedure.
- A variable-length character string reserved for future use.
- A variable-length character string containing 80 blanks on input, and an 80-byte or shorter error message on output. This error message is placed in the SQLCA and used as the SQL error message for the CALL PROCEDURE statement.
- A two-byte fixed-length character string containing the CA Datacom/DB external error code on output.
- A single character containing the CA Datacom/DB internal error code on output.

The logic inside SQL for parameter style DATACOM SQL is as follows:

When SQL gains control after a successful execution of the procedure, that is, after the procedure code loaded, ran, and did not abend, *the internal SQL code and error message are reset* to the values returned in the user's parameter list, regardless of whether it was triggered or called, even if this means a non-zero SQLCODE becomes zero or is replaced.

In order to minimize the confusion that a newly-defined trigger can cause for a pre-existing application that uses the navigational (record-at-a-time) API rather than SQL, we handle the DB return codes for DATACOM SQL style procedures as follows.

If the SQLCODE returned from the procedure is zero or positive, a non-blank CA Datacom/DB return code is ignored. If the procedure returned a negative SQLCODE and was explicitly called (as opposed to being triggered), the CA Datacom/DB external and internal return codes are reset to the values returned through the procedure's parameter list. If the procedure returned a negative SQLCODE and was triggered, we store the SQLCODE at offset 26 decimal (signed binary fullword) into the user's Request Area, force the CA Datacom/DB return code(s) to 94(100), then document the CA Datacom/DB return codes returned from the procedure at offsets 30 decimal (2 byte character) and 32 decimal (one byte unsigned binary) into the user's Request Area for the external and internal return codes, respectively. This is done to allow users of navigational programs to differentiate between failures inside procedures and those related to their specific CA Datacom/DB requests, because they generally do not have logic to interpret an SQLCODE.

You have complete control (and responsibility) in deciding whether what occurs constitutes a *success.* You must set the SQLCODE and error message parameters on exit in one of these three ways:

- If for some reason you want to fail even if all SQL requests received an SQLCODE=0, set SQLCODE -534 and provide an error message containing 80 bytes as desired, or
- Supply the SQLCODE, error message, and CA Datacom/DB external and internal return codes *exactly as SQL returned it to the procedure,* or
- Pass back an SQLCODE forced to 0 at your discretion.

SQL

When a procedure is created using PARAMETER STYLE DATACOM SQL in the CREATE PROCEDURE statement, the SQLSTATE status indicator is returned in the SQLCA. For detailed information about the SQLSTATE status indicator, see SQL States.

Parameter style SQL passes nulls to the procedure as does GENERAL WITH NULLS and DATACOM SQL. It also passes these additional four parameters that are added to the end of the parameter/null indicator list:

- The SQLSTATE (INOUT, but always passed in as 00000, similar to the SQLCODE in style DATACOM SQL, that is, it is passed to the procedure as 00000 and used to set the SQLSTATE of the CALL PROCEDURE statement on output).
- Authid.procedure-name (IN, same as in style DATACOM SQL, that is a variable-length character string containing the name of the procedure).
- Authid.specific name (IN, same as in style DATACOM SQL, that is, a variable-length character string reserved for future use).
- Error message text (INOUT, passed in as 0-length string, same as DATACOM SQL, that is, a variable-length character string containing 80 blanks on input, and an 80-byte or shorter error message on output that is placed in the SQLCA and used as the SQL error message for the CALL PROCEDURE statement).

Unlike style DATACOM SQL, the CA Datacom/DB external and internal return codes are not a part of this parameter list but are encoded in the generated SQLSTATE value. For example, the SQLSTATE that equates to SQL return code -117 is Seeii, and the SQLSTATE that equates to SQL return code -118 is Reeii, where ee represents the 2-byte external CA Datacom/DB return code, and ii is the CA Datacom/DB internal return code in hexadecimal characters.

As an aid in understanding error recovery for procedures, note the following:

- The PARAMETER STYLE being used by the procedure which is executing at each level of nesting controls the SQLCODE seen by the logic that called it, if procedures execute in a nested fashion due either to:
  - CALL PROCEDURE statements inside procedures, or
  - TRIGGERs triggering during execution of a procedure.

  This means that the PARAMETER STYLE of any procedure controls how the success or failure of everything that occurred during execution of the procedure is interpreted, including recursively called procedures whose PARAMETER STYLEs might not match that of the *outermost* procedure.

- As with all SQL statements, if an SQL statement is issued from a procedure and fails, it has no effect on the database. This rule holds true at every level of recursion. For example, if an INSERT statement, issued by a procedure, triggers procedure calls five layers deep and results in the updating of 500 rows in the database but then fails, not only is the INSERT backed out, the 500 updates that executed during processing of the INSERT are also backed out. Even though hundreds of SQL statements have been rolled back, at the level of the procedure that executed the INSERT, only one statement was backed out. Limited-scope rollbacks such as this (that occur automatically in lower levels of recursion) in no way affect the ability of the higher levels of procedures to either continue processing or to abort and return errors to callers, triggering additional automatic rollbacks as needed. Note, however, that users are not allowed to code their own ROLLBACK or COMMIT statements inside procedures.

SQL Error Messages Related to Procedures and Triggers

The following SQL error codes have been added in support of procedures and triggers:

-321

INVALID SQLCODE sqlcode HAS BEEN GENERATED

Reason:

A user-written procedure has returned an SQLCODE that is not a valid DATACOM SQLCODE.

The SQLSTATE that equates to this SQL return code is 39001.

Action:

Modify, repreprocess, and recompile the procedure to follow all instructions given in Parameter Styles and Error Handling (see page 66).

-530

PROC authid.name: msg-string

Reason:

There has been a procedure preparation error. The information in msg-string varies depending upon the error that has occurred.

The SQLSTATE that equates to this SQL return code is 38S01.

Action:

Correct the problem described by the msg-string.

-531

PROC authid.name: msg-string

Reason:

There has been a procedure execution error. The information in msg-string varies depending upon the error that has occurred. This message commonly occurs when you have not concatenated the load library (into which your procedure has been linked) into the STEPLIB of the DBMUFPR of your Multi-User Facility job. This is especially likely to be the cause of the message if the message resembles the following:

PROC authid.sql-proc-name: external-proc-name FETCH ERROR

For example, **PROC SYSUSR.MYPROC: MYPROC FETCH ERROR.**

The SQLSTATE that equates to this SQL return code is 38S02.

Action:

Correct the problem described by the msg-string.

-532

TRIG authid.name: msg-string

Reason:

There has been a trigger preparation error. The information in msg-string varies depending upon the error that has occurred.

The SQLSTATE that equates to this SQL return code is 09S02.

Action:

Correct the problem described by the msg-string.

-533

TRIG authid.name: msg-string

Reason:

There has been a trigger execution error. The information in msg-string varies depending upon the error that has occurred. This message commonly occurs when you have not concatenated the load library (into which your procedure has been linked) into the STEPLIB of the DBMUFPR of your Multi-User Facility job. This is especially likely to be the cause of the message if the message resembles the following:

TRIG authid.sql-trig-name: external-proc-name FETCH ERROR

For example, **TRIG SYSUSR.MYTRIG: MYPROC FETCH ERROR.**

The SQLSTATE that equates to this SQL return code is 09S01.

Action:

Correct the problem described by the msg-string.

-534

msg-string

Reason:

There has been a user-defined procedure execution error. This SQL error code only occurs in procedures whose parameter style is SQL or DATACOM SQL. The information in the msg-string varies depending upon the error that has occurred, that is to say, user-written procedure logic creates the entire error message. The message is truncated if it exceeds the 80-byte length of the SQLCA error message area.

The SQLSTATE that equates to this SQL return code is 2FS04

Action:

Correct the problem described by the msg-string.

-535

PROC authid.name: msg-string

Reason:

There has been an environmental problem, possibly LE-related, that prevented the procedure from running. In the message, authid.name identifies the PROC and msg-string specifies the cause.

The SQLSTATE that equates to this SQL return code is 39S01.

Action:

Correct the problem described by the msg-string.

-537

msg-string

Reason:

There has been a user-defined execution error in an SQL procedure (a "LANGUAGE SQL" procedure). The information in the msg-string varies, depending upon the error that has occurred, that is to say, user-written procedure logic creates the entire error message. The message is truncated if it exceeds the 80-byte length of the SQLCA error message area.

The SQLSTATE that equates to this SQL return code is 38S04.

Action:

Correct the problem described by the msg-string.

Datadictionary Support for Triggers and Procedures

Beginning in r10, implementation of CA Datacom Datadictionary support for triggers and procedures involved the addition of new entity-types, attributes, and relationships to the model used in the previous Version. The following page contains a diagrammatic view of the changes to the previous model.

Processing

Following is described the requirements of the SQL/Datadictionary interface for each of the SQL statements affected by triggers and procedures.

ALTER TABLE Processing Modifications

When an ALTER TABLE statement is processed for a table with any triggers with Event Times of either Before Update or After Update and a Column Dependency, these triggers are marked for Automatic Rebind by setting the Trigger Valid Indicator field of the Trigger DDD Member to N.

If an ALTER TABLE attempts to delete a Column on which a trigger depends, the ALTER fails with a DSF Return Code of TUC.

COMMENT ON Processing

The parameters and processing are the same as the current COMMENT ON requests for tables, views, and synonyms.

DROP TABLE Processing Modifications

When a Table is dropped, all TRIGGER occurrences and their respective DDD Members referring to the Table in the *on table* SQL clause are deleted.

Examples: Creating a Procedure

This section takes you through the process of creating a procedure as follows:

1. Coding the Program (see following)
2. Defining the Procedure to SQL (see Defining the Procedure to SQL (see page 91))

Coding the Program

Consider an application that updates rows in a table representing a catalog of auto parts that can be ordered over the Internet, requiring the ability to update the catalog in real time. This capability requires a complex series of transactions. Decisions must be made during processing. A cascading foreign key alone cannot satisfy these needs. A procedure is the most efficient way to fulfill the requirements.

Sample JCL for C

Following is an example of coding the needed procedure in C. The comments in the procedure program example provide a guide to the procedure building process.

For COBOL examples see Sample JCL for z/OS (see page 82) for z/OS and Sample JCL for z/VSE for z/VSE. PL/I or Assembler could also have been used to code the procedure.

Before coding your first procedure, see Transaction Integrity (see page 61), Restrictions (see page 62), and Parameter Styles and Error Handling (see page 66).

Note: Use the following as a guide to prepare your JCL. The JCL statements are for example only. Lowercase letters in a statement indicate a value you must supply. Code all statements to your site and installation standards.

```
//jobname     See the note above.
000002 //            CLASS=A,MSGCLASS=X,REGION=4M
000003 //TWOUP  OUTPUT DEFAULT=YES,FORMDEF=010111,PAGEDEF=W120C0,CHARS=(GT20)
000004 //JOBLIB   DD DSN=DCMDEV.DB.MUF3.LODLIB,DISP=SHR
000005 //         DD DSN=dsnname.LODLIB,DISP=SHR
000006 //         DD DSN=dsnname.LODLIB2,DISP=SHR
000007 +INC QE.LIBS10X  GET THE REST OF THE 10.0 LIBS
000008 //         DD DSN=IBMPROD.CEE.SCEERUN.IGZEPSX,DISP=SHR
000009 //         DD DSN=CEE.SCEERUN,DISP=SHR
000010 //         DD DSN=CEE.AIGZMOD1,DISP=SHR
000011 //         DD DSN=DCMDEV.SQL.TOOLS.LODLIB,DISP=SHR
000012 +INC GRB.EDCC (OS 390 C/C++ COMPILE PROC)
000013 //* ****************************
000014 //* *  "C" PRECOMPILE STEP   ***
000015 //* ****************************
000016 //CPRECMP  EXEC PGM=DBPLIPR,PARM='PLANNAME=ITEMKILL'
000017 //PROCLIB  DD DSN=CA90SMVS.NEWC.R3V1.PROCLIB,DISP=SHR
000018 //*
000019 //*   The PROCSQLUSAGE option below identifies this program as a procedure.
000020 //*
000021 //OPTIONS  DD *
000022 PROCSQLUSAGE=MODIFIES
000023 LANG=C
000024 SQLMODE=DATACOM
000025 AUTHID=SYSADM
000026 ISOLEVEL=C
000027 DATE=JIS
000028 /*
000029 //SOURCE   DD *
000029
```

```
        /**********  Program source starts below.        ************/
                  /* ITEMKILL - C Procedure example. */
/*
**  This procedure is triggered when a supplier cancels production of
**  a product in our consumer catalog.  The program checks to see how
**  many open orders we need to cancel and decides, based on this
**  number, whether to send apology letters to a small number of
**  customers, or to generate an error message instructing us to contact
**  the supplier to attempt to fill the orders.  This procedure is
**  passed an input parameter that determines the number of orders we
**  are willing to cancel (if any).
*/

/* The procedure you write must be re-entrant. */
#pragma options(RENT)
/*
**  Use of the linkage pragma is required to tell the C compiler that
**  our load module is "fetched" for execution at runtime.
*/
#pragma linkage(itemKill,FETCHABLE)

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
/*  The following structure maps to a VARCHAR(128) data item in SQL (FYI). */
typedef struct varChar128

   short length;
   char  data 128á;
   SQL_VARCHAR_128;
/*
** The following structure maps to the additional parameters passed
** to your procedure when the "DATACOM SQL" parameter-style has
** been specified by the CREATE PROCEDURE.  Note that the variable
** containing the sqlcode may not be named "sqlcode" because our
** precompiler generates an SQLCA that uses the name.  These parameters
** enable your program to control the SQLCODE that SQL sees as
** the result of the CALL/EXECUTE PROCEDURE statement that
** was executed or triggered.  Note that a negative SQLCODE-OUT
** aborts any INSERT, UPDATE, or DELETE that triggers it.  See the
** "Parameter Styles and Error Handling" section for more details.
*/
```

```
typedef struct parmsDatacomSQL

   int             *sqlcodeOut;
   SQL_VARCHAR_128 *procName;
   SQL_VARCHAR_128 *specName;
   SQL_VARCHAR_128 *errMsgOut; /* Truncated to 80 bytes in 10.0. */
   char            *dbExtCodeOut;
   short           *dbIntCodeOut;
   SQL_PROC_PARMS_DCM;
void userDefinedErrorDoc(SQL_PROC_PARMS_DCM *dcmSqlParms, char *errMsg);
/*
** The function name used below must match both that of the load-
** module that we are going to produce, and the EXTERNAL name defined
** by the CREATE PROCEDURE statement that we execute later.
**
** Note that the data pointed to by the formal parameters
** precisely correspond to the parameter definitions specified
** in the CREATE PROCEDURE statement and appear in the
** same order.  The C-language variables chosen to process the
** data must match in data-type, size, and order.
**
** When SQL regains control after execution of the procedure, it
** ignores any data that your program stored into parameters defined
** by the CREATE PROCEDURE statement to be input only ("IN").
**
** In order to minimize the confusion that a newly-defined trigger
** can cause for a preexisting application that uses the navigational
** (record-at-a-time) API rather than SQL, we handle DB return codes
** for DATACOM SQL style procedures as follows:
**
** If the SQLCODE returned from the procedure is zero or positive, a
** nonblank DB return code is ignored.  If the procedure returned a
** negative SQLCODE and was explicitly called as opposed to being
** triggered, DB external and internal return codes are reset to the
** values returned through the procedure's parameter list.  If the
** procedure returned a negative SQLCODE and was triggered, we store
** the SQLCODE at offset 26 decimal (signed binary fullword) into the
** user's Request Area, for the DB return code(s) to 94(100), then
** then document the DB return codes returned from the procedure
** at offsets 30 decimal (2-byte character) and 32 decimal (1-byte
** unsigned binary) into the user's Request Area for the external and
** internal return codes, respectively.  This is done to allow users
** of navigational programs to differentiate between failures
** inside procedures and those related to their specific DB requests,
** since they generally do not have logic to interpret an SQLCODE.
**
```

```
** Note that the first three parameters would appear in the formal
** parameter list regardless of the PARAMETER STYLE specified
** by the CREATE PROCEDURE statement.  The next three parameters
** are null indicator variables corresponding to the first three,
** and appear only under certain parameter styles (in Version
** 10.0, they appear under DATACOM SQL and GENERAL WITH NULLS).
** The "dcmSqlParms" parameter (see previous explanation) appears
** only under parameter style DATACOM SQL.
*/
int itemKill(int   *canceledPartIdIn,  int   *vendorIdIn,
             int   *maxBadOrdersIn,    short *canceledPartIdNull,
             short *vendorIdNull,      short *maxBadOrdersNull,
             SQL_PROC_PARMS_DCM dcmSqlParms)


   EXEC SQL BEGIN DECLARE SECTION;
   int   canceledPartId     = *canceledPartIdIn;
   int   vendorId           = *vendorIdIn;
   int   numBackOrders      = 0;
   int   numOrdersCanceled  = 0;
   char *errMsg;
   EXEC SQL END DECLARE SECTION;

   EXEC SQL WHENEVER NOT FOUND goto end;
   EXEC SQL WHENEVER SQLERROR  goto sqlError;
   /*
   ** Initialize output parameters.
   ** Since triggers may not call procedures that have output
   ** (OUT or INOUT) parameters, and we intend to use this
   ** procedure as a trigger, we have coded/created it without
   ** output parameters other than those required for parameter
   ** style DATACOM SQL.
   */
```

```
*(dcmSqlParms.sqlcodeOut)     = 0;
*(dcmSqlParms.dbExtCodeOut)   = 0;
*(dcmSqlParms.dbIntCodeOut)   = 0;
dcmSqlParms.errMsgOut->length = 0;  /* SQL 10.0 maximum is 80. */

memset(dcmSqlParms.errMsgOut->data, 0, 80);

/* Handle nulls on input. */
if (*canceledPartIdNull == -1)
   errMsg = "ITEM_ORDER_KILLER ABORTED: CANCELED PART ID IS NULL";
else if (*vendorIdNull == -1)
   errMsg = "ITEM_ORDER_KILLER ABORTED: VENDOR ID IS NULL";
else

   errMsg = NULL;
   if (*maxBadOrdersNull == -1)
      *maxBadOrdersIn = 0;


/* Quit if an error message was produced. */
if (errMsg)

   userDefinedErrorDoc(&dcmSqlParms., errMsg);
   goto end;
/* How many back orders for this item have to be canceled? */

EXEC SQL
   select count(*)
   into   :numBackOrders
   from   sales.order_items
   where  item_id     = :canceledPartId and
          item_status = 'BACK-ORDERED';

/* Handle outstanding orders. */
if (numBackOrders > 0)

   /*
   ** This cancellation by the supplier affects too many
   ** orders.  Try to get him to honor the orders.
   */
   if (numBackOrders > *maxBadOrdersIn)
      userDefinedErrorDoc(&dcmSqlParms.,
      "ITEM_ORDER_KILLER DETECTED EXCEEDED ORDER CANCELLATION LIMIT");
   else

      /*
      ** Cancel orders and send apology letters to customers
      ** whose orders are being canceled.
      */
```

```
            EXEC SQL
               update sales.order_items
               set    item_status = 'CANCELED',
                      comments    = 'ITEM DISCONTINUED'
               where  item_id     = :canceledPartId and
                      item_status = 'BACK-ORDERED';

            EXEC SQL
               insert into customer.apology_letters
                      (customer_id, order_id, item_id, quantity,
                       comments, problem_type)
               select A.customer_id,  A.order_id, B.item_id,
                      B.quantity, B.comments, 'ITEM DISCONTINUED'
               from   sales.orders A, sales.order_items B
               where  A.order_id    = B.order_id       and
                      B.item_id     = :canceledPartId and
                      B.item_status = 'CANCELED';

            numOrdersCanceled = numBackOrders;
        /* Record the problem so we can track "problem" vendors. */
        EXEC SQL
           insert into vendor.problems
                  (vendor_id, problem_type, num_orders_affected,
                   num_orders_canceled, related_item_id,
                   problem_date, resolution_date)
           values (:vendorId, 'ITEM DISCONTINUED', :numBackOrders,
                   :numOrdersCanceled, :canceledPartId,
                   CURRENT DATE, NULL);
     end:
        return(0);
    sqlError:
        /*
        ** Supply error information to caller using output parameters.
        ** Note that the precompiler automatically includes the "sqlca"
        ** structure in your program.
        */
        *(dcmSqlParms.sqlcodeOut)      = sqlca.sqlca_code;
        *(dcmSqlParms.dbIntCodeOut)    = sqlca.sqlca_dbcode_int;
        dcmSqlParms.errMsgOut->length  = sqlca.sqlca_err_len;
        memcpy(dcmSqlParms.dbExtCodeOut,    sqlca.sqlca_dbcode_ext, 2);
        memcpy(dcmSqlParms.errMsgOut->data, sqlca.sqlca_err_msg,
               sqlca.sqlca_err_len);
        /*
```

```
    ** Note that the output of this "printf" statement would have
    ** appeared in a SYSOUT file attached to the output of the
    ** Multi-user job, so I have decided its use here is inappropriate:
    ** printf("ITEMKILL FAILED WITH SQLCODE = %d.", *(dcmSqlParms.sqlcodeOut));
    */

    goto end;

 /* Generate documentation for a user-defined error. */
void userDefinedErrorDoc(SQL_PROC_PARMS_DCM *dcmSqlParms, char *errMsg)

    *(dcmSqlParms->sqlcodeOut)     = -534;/* User-defined error. */
    dcmSqlParms->errMsgOut->length = 80;  /* SQL 10.0 maximum. */

    memcpy(dcmSqlParms->errMsgOut->data, errMsg,
           (strlen(errMsg) > 80 ? 80 : strlen(errMsg)));
    return;
000200 /*
000201 //SYSUDUMP DD SYSOUT=*
000202 //REPORT   DD SYSOUT=*
000203 //SYSPRINT DD SYSOUT=*
000204 //SYSOUT   DD SYSOUT=*
000205 //SRCOUT   DD DSN=&.&SRC.,DISP=(,PASS,DELETE),UNIT=VIO,
000206 //            SPACE=(2000,(200,200)),
000207 //            DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
000208 //*
000209 //* LE C COMPILE
000210 //COMPA    EXEC PROC=EDCC,        (FROM SYS2.PROCLIB)
000211 //   CRUN='RENT',
000212 //   CPARM='NOMARGINS,NOSEQUENCE,LIST,SOURCE',
000213 //   CPARM2='LOCALE("POSIX"),LANGLVL(ANSI),OMVS,DLL',
000214 //   CPARM3='SSCOM,LONGNAME,SHOWINC',
000215 //  INFILE='&.&SRC'.,
000216 //  OUTFILE='dsnname.OBJLIB(ITEMKILO)'
000217 //*
000218 //SYSLIB   DD  DSN=CEE.SCEEH.H,DISP=SHR
000219 //         DD  DSN=CEE.SCEEH.SYS.H,DISP=SHR
000220 //USERLIB  DD  DSN=DCMDEV.SQL.LIBRMAST,DISP=SHR,SUBSYS=LAM
000221 //* *********** LINK STARTS HERE **********************
000222 //PRELINK   EXEC  PGM=EDCPRLK,
000223 //  PARM='POSIX(OFF)/OE,MEMORY,DUP,NOER,MAP,NOUPCASE,NONCAL'
```

```
000224 //*
000225 //SYSMSGS  DD  DSN=CEE.SCEEMSGP(EDCPMSGE),DISP=SHR
000226 //OBJLIB   DD  DSN=dsnname.OBJLIB,DISP=SHR
000227 //C8941    DD  DSN=CEE.SCEEOBJ,DISP=SHR
000228 //SYSOUT   DD  SYSOUT=*
000229 //SYSPRINT DD  *
000230 //SYSMOD   DD  DSN=dsnname.OBJLIB(ITEMKOBJ),DISP=SHR
000231 //SYSDEFSD DD  DUMMY
000232 //SYSIN    DD  *
000233   INCLUDE OBJLIB(ITEMKILO)
000234 /*
000235 //LINKEDIT  EXEC  PGM=LINKEDIT,
000236 //  PARM=('AMODE=31,RMODE=ANY,TERM=YES,MSGLEVEL=0,MAP,DYNAM=DLL',
000237 //        'CALL=YES,CASE=MIXED,REUS=RENT,EDIT=YES')
000238 //SYSLIB   DD  DSN=CEE.SCEELKED,DISP=SHR
000239 //         DD  DSN=SYS1.CSSLIB,DISP=SHR
000240 //SYSPRINT DD  SYSOUT=*
000241 //SYSTERM  DD  SYSOUT=*
000242 //SYSLMOD  DD  DISP=SHR,DSN=dsnname.LODLIB2(ITEMKILL)
000243 //OBJLIB   DD  DSN=dsnname.OBJLIB,DISP=SHR
000244 //CALIB    DD  DSN=DCMDEV.DB.R100.LODLIB,DISP=SHR
000245 //CEELIB   DD  DSN=DCMDEV.DBDT.DSYTEST.LOADLIB,DISP=SHR
000246 //SYSLIN   DD  *
000247   INCLUDE  OBJLIB(ITEMKOBJ)
000248   INCLUDE  CEELIB(CEEUOPT)
000249   INCLUDE  CALIB(DBXPIPR)
000250 /*
```

Please note that the library containing the procedure's load module must be added to the STEPLIB or JOBLIB of the Multi-User Facility startup JCL.

Sample JCL for z/OS

Following is the z/OS COBOL functional equivalent of the C procedure shown previously (see Sample JCL for C (see page 74)). For a z/VSE sample in COBOL, see Sample JCL for z/VSE.

The following JCL example is for z/OS sites.

Note: Use the following as a guide to prepare your JCL. The JCL statements are for example only. Lowercase letters in a statement indicate a value you must supply. Code all statements to your site and installation standards.

```
//jobname     See the note above.
000001 //            CLASS=A,MSGCLASS=X,REGION=2048K
000002 //JOBLIB   DD DSN=library-containing-DBSIDPR,DISP=SHR
000003 //         DD DSN=library-containing-multi-user-modules,DISP=SHR
000004 //         DD DSN=etc...,DISP=SHR
000005 //*--------------------------------------------------------------------*
000006 //* STEP 1: PRE-COMPILE THE PROCEDURE PROGRAM                  *
000007 //*--------------------------------------------------------------------*
000008 //PRECOMP  EXEC PGM=DBXMMPR
000009 //WORK1    DD   DSN=&.&WORK1.,UNIT=SYSDA,DISP=(NEW,PASS),
000010 //         DCB=(RECFM=F,LRECL=80,BLKSIZE=80),SPACE=(TRK,(1,1))
000011 //WORK2    DD   DSN=&.&WORK2.,UNIT=SYSDA,DISP=(NEW,PASS),
000012 //         DCB=(RECFM=F,LRECL=80,BLKSIZE=80),SPACE=(TRK,(1,1))
000013 //WORK3    DD   DSN=&.&WORK3.,UNIT=SYSDA,DISP=(NEW,PASS),
000014 //         DCB=(RECFM=F,LRECL=80,BLKSIZE=80),SPACE=(TRK,(1,1))
000015 //SYSOUT   DD   SYSOUT=*
000016 //SYSPRINT DD   SYSOUT=*
000017 //SNAPER   DD   SYSOUT=*
000018 //SNAPER   DD   SYSOUT=*
000019 //SYSPUNCH DD   DSN=&.&SQLCOB.,UNIT=SYSDA,DISP=(NEW,PASS),
000020 //         DCB=(RECF  M=F,LRECL=80,BLKSIZE=80),SPACE=(TRK,(1,1))
000021 //SYSUDUMP DD   SYSOUT=*
000022 //SYSIN    DD   *
       ***           ITEMKILL - COBOL Procedure example             ***
       *           (Line numbers removed for clarity)
       * This procedure is triggered when a supplier cancels production
       * of a product in our consumer catalog. The program checks to see
       * how many open orders we need to cancel and decides, based
       * on this number, whether to send apology letters to a small
       * number of customers, or to generate an error message instructing
       * us to contact the supplier to attempt to fill the orders.
       * This procedure is passed an input parameter that determines the
       * the number of orders we are willing to cancel (if any).

       *  The "PROCSQLUSAGE" option used below identifies this program
       *  as a procedure.
```

```
*$DBSQLOPT PROCSQLUSAGE=MODIFIES USRNTRY=NONE
*$DBSQLOPT SQLMODE=DATACOM AUTHID=SYSADM ISOLEVEL=C

 IDENTIFICATION DIVISION.

* The PROGRAM-ID must match both the name of the load-module
* that we are going to produce, and the EXTERNAL name defined by
* the CREATE PROCEDURE statement that we execute later.

 PROGRAM-ID. ITEMKILL.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. IBM-370.
 OBJECT-COMPUTER. IBM-370.
 INPUT-OUTPUT SECTION.
 DATA DIVISION.

 WORKING-STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC
 01  NUM-BACK-ORDERS        PIC S9(9) COMP.
 01  NUM-ORDERS-CANCELED    PIC S9(9) COMP VALUE 0.
     EXEC SQL END   DECLARE SECTION END-EXEC.

 LINKAGE SECTION.
***  DECLARE PROCEDURE PARAMETERS
*    The variables declared in the linkage section must
*    precisely match the parameter definitions specified
*    in the CREATE PROCEDURE statement, and must appear
*    in the same order.  When SQL regains control after
*    execution of the procedure, it ignores data that
*    your program stored into parameters defined by the
*    CREATE PROCEDURE statement to be input only ("IN").

*    See the "Parameter Style and Error Handling" section
*    for more details.

 01  CANCELED-PART-ID-IN    PIC S9(9) COMP.
 01  VENDOR-ID-IN           PIC S9(9) COMP.
 01  MAX-BAD-ORDERS-IN      PIC S9(9) COMP.
***  DECLARE NULL INDICATORS FOR THE PARAMETERS.
*    These declarations must be included if (and only if) the
*    PARAMETER STYLE specified in your CREATE PROCEDURE is
*    GENERAL WITH NULLS or DATACOM SQL.

 01  CANCELED-PART-ID-NULL  PIC S9(4) COMP.
 01  VENDOR-ID-NULL         PIC S9(4) COMP.
 01  MAX-BAD-ORDERS-NULL    PIC S9(4) COMP.
```

```
***  DECLARE "PARAMETER STYLE DATACOM SQL" OUTPUT PARAMETERS
*    These declarations must be included if (and only if) your
*    PARAMETER STYLE is DATACOM SQL.  A copybook containing
*    these declarations is provided with CA Datacom/DB SQL.
*    The variable containing the sqlcode may not be named
*    "SQLCODE" since the SQLCA uses this name.
*    These additional parameters allow the procedure
*    to control the SQLCODE that SQL sees as the
*    result of the CALL/EXECUTE PROCEDURE statement that
*    was executed or triggered.  Note that a negative
*    SQLCODE-OUT aborts any INSERT, UPDATE, or DELETE
*    that triggers it.

*    See the "Parameter Styles and Error Handling" section
*    for more details on this.

 01  SQLCODE-OUT      PIC S9(9) COMP.
 01  PROCNAME.
     49 PROCNAME-LEN  PIC S9(4) COMP.
     49 PROCNAME-TEXT PIC X(128).
 01  SPECNAME.
     49 SPECNAME-LEN  PIC S9(4) COMP.
     49 SPECNAME-TEXT PIC X(128).
*    Note that in SQL 10.0, error messages longer than 80 bytes
*    are truncated.  In the future, this may not be the case.

 01  ERRMSG.
     49 ERRMSG-LEN    PIC S9(4) COMP.
     49 ERRMSG-TEXT   PIC X(128).
 01  DBCODE-EXT       PIC X(2).
 01  DBCODE-INT       PIC S9(4) COMP.

 PROCEDURE DIVISION USING CANCELED-PART-ID-IN,   VENDOR-ID-IN,
     MAX-BAD-ORDERS-IN,   CANCELED-PART-ID-NULL, VENDOR-ID-NULL,
     MAX-BAD-ORDERS-NULL,
     SQLCODE-OUT, PROCNAME, SPECNAME, ERRMSG, DBCODE-EXT,
     DBCODE-INT.
     EXEC SQL WHENEVER SQLERROR   GO TO SQL-ERROR-RTN END-EXEC.
     EXEC SQL WHENEVER NOT FOUND  CONTINUE            END-EXEC.
```

```
*** Initialize values of output parameters for SQL.
*   Since triggers may not call procedures that have output
*   (OUT or INOUT) parameters, and we intend to use this
*   procedure as a trigger, we have coded/created it without
*   output parameters other than those required for parameter
*   style DATACOM SQL.

     MOVE 0          TO SQLCODE-OUT.
     MOVE 0          TO ERRMSG-LEN.
     MOVE 'NO ERROR' TO ERRMSG-TEXT.
     MOVE '  '       TO DBCODE-EXT.
     MOVE 0          TO DBCODE-INT.
*    Handle nulls on input.

     IF (CANCELED-PART-ID-NULL = -1)
        MOVE
           'ITEM_ORDER_KILLER ABORTED: CANCELED PART ID IS NULL'
              TO ERRMSG-TEXT
        GO TO USER-DEFINED-ERROR
     ELSE IF (VENDOR-ID-NULL = -1)
        MOVE 'ITEM_ORDER_KILLER ABORTED: VENDOR ID IS NULL'
              TO ERRMSG-TEXT
        GO TO USER-DEFINED-ERROR
     ELSE IF (MAX-BAD-ORDERS-NULL = -1)
        MOVE 0 TO MAX-BAD-ORDERS-IN.
*    How many back orders for this item are affected?

     EXEC SQL
         select count(*)
         into   :NUM-BACK-ORDERS
         from   sales.order_items
         where  item_id     = :CANCELED-PART-ID-IN and
                item_status = 'BACK-ORDERED';
     END-EXEC.

*    Handle outstanding orders.
```

```
     IF (NUM-BACK-ORDERS > 0)

*        This cancellation by the supplier affects too many
*        orders.  Try to get him to honor the orders.

         IF (NUM-BACK-ORDERS > MAX-BAD-ORDERS-IN)

*            Generate a user-defined error.

             MOVE
             "ITEM_ORDER_KILLER FOUND EXCEEDED ORDER CANCELLATION LIMIT"
                TO ERRMSG-TEXT
         ELSE

*            Mark orders and send apology letters to customers
*            whose orders are being canceled.

             EXEC SQL
                 update sales.order_items
                 set    item_status = 'CANCELED',
                        comments    = 'ITEM DISCONTINUED'
                 where  item_id     = :CANCELED-PART-ID-IN and
                        item_status = 'BACK-ORDERED';
             END-EXEC
             EXEC SQL
                 insert into customer.apology_letters
                        (customer_id, order_id, item_id, quantity,
                         comments, problem_type)
                 select A.customer_id,  A.order_id, B.item_id,
                        B.quantity, B.comments,'ITEM DISCONTINUED'
                 from   sales.orders A, sales.order_items B
                 where  A.order_id    = B.order_id       and
                        B.item_id     = :CANCELED-PART-ID-IN and
                        B.item_status = :'CANCELED';
             END-EXEC
```

```
            MOVE NUM-BACK-ORDERS TO NUM-ORDERS-CANCELED.
*        Record the problem so we can track "problem" vendors.

         EXEC SQL
             Insert into vendor.problems
                    (vendor_id, problem_type, num_orders_affected,
                     num_orders_canceled, related_item_id,
                     problem_date, resolution_date)
             values (:VENDOR-ID-IN,'ITEM DISCONTINUED',
                     :NUM-BACK-ORDERS, :NUM-ORDERS-CANCELED,
                     :CANCELED-PART-ID-IN, CURRENT DATE, NULL);
         END-EXEC.

*    Does ERRMSG-TEXT indicate a user-defined error occurred?

     IF (ERRMSG-TEXT NOT EQUAL 'NO ERROR')
        GO TO USER-DEFINED-ERROR.

     GOBACK.
*** Supply error information to output parameters.
*   Copy the error diagnostics we received from SQL in our
*   SQLCA to the output parameters, which in turn are
*   copied by SQL into the SQLCA of the calling CALL/EXECUTE
*   PROCEDURE statement.

 SQL-ERROR-RTN.
     MOVE SQLCA-ERR-MSG    TO ERRMSG-TEXT.
     MOVE SQLCA-DBCODE-EXT TO DBCODE-EXT.
     MOVE SQLCA-DBCODE-INT TO DBCODE-INT.
     MOVE SQLCODE          TO SQLCODE-OUT.
     MOVE 80               TO ERRMSG-LEN.
*** Note that the output of this display statement would have appeared
*   in a SYSOUT file attached to output of the Multi-User job, so
*   I have decided its use is inappropriate.
*   DISPLAY 'SQLCODE =' SQLCODE-OUT'

    GOBACK.

 USER-DEFINED-ERROR.
*    ERRMSG-TEXT has already been set.
     MOVE 80               TO ERRMSG-LEN.
     MOVE -534             TO SQLCODE-OUT.
     GOBACK.

****   End of Program.  JCL continues below.          ******
```

```
000260 /*
000261 //*       *** End of Program and Continuation of JCL. ***
000262 //*
000263
//*--------------------------------------------------------------------*
000264 //* STEP2: COMPILE COBOL USER PROGRAM OUTPUT FROM COBOL PRECOMPILER   *
000265
//*--------------------------------------------------------------------*
000266 //*
000267 //COBOL  EXEC PGM=IGYCRCTL,
000268 //            PARM='RENT,NUM,NODYN,APOST,NOSEQUENCE,LIST',
000269 //            COND=(8,LT)
000270 //SYSLIN   DD  DISP=(MOD,PASS),DSN=&.&COBOLOD.,
000271 //            UNIT=SYSDA,SPACE=(TRK,(15,15))
000272 //SYSPRINT DD  SYSOUT=*
000273 //SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
000274 //SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
000275 //SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
000276 //SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
000277 //SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
000278 //SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
000279 //SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
000280 //SYSIN    DD  DSN=&.&SQLCOB.,UNIT=SYSDA,DISP=(OLD,DELETE,DELETE)
000281 /*
000282
```

```
 //*-------------------------------------------------------------------*
 000283 //* STEP3: LINK USER PROGRAM WITH SYSTEM MODULES
*
 000284
 //*-------------------------------------------------------------------*
 000285 //*
 000286 //LINK   EXEC LKED,COND=(8,LT),
 000287 //            PARM.LKED='RENT,XREF,LIST,LET,MAP'
 000288 //LKED.SYSLIN DD DSN=&.&COBOLOD.,UNIT=SYSDA,DISP=(OLD,DELETE,DELETE)
 000289 //            DD DDNAME=SYSIN
 000290 //LKED.SYSLMOD DD DSN=dsnname.LODLIB2,DISP=SHR
 000291 //*KED.SYSLIB DD DSN=MVSSYS.COB2.V1R3M2.COB2LIB,DISP=SHR
 000292 //LKED.SYSLIB DD DSN=CEE.SCEELKED,DISP=SHR
 000293 //LKED.OBJLIB DD DSN=DCMALL.R900.CAILIB,DISP=SHR
 000294 //            DD DSN=DCMDEV.DB.R100.TST.LODLIB,DISP=SHR
 000295 //            DD DSN=DCMDEV.DB.R100.LODLIB,DISP=SHR
 000296 //LKED.CEELIB DD DSN=DCMDEV.DBDT.DSYTEST.LOADLIB,DISP=SHR
 000297 //LKED.SYSIN  DD *
 000298  INCLUDE CEELIB(CEEUOPT)
 000299  INCLUDE OBJLIB(DBXHVPR)
 000300  INCLUDE OBJLIB(DBXPIPR)
 000301  NAME ITEMKILL(R)
 000302 /*
```

Note that the library containing the procedure's load module must be added to the STEPLIB or JOBLIB of the Multi-User Facility startup JCL.

Sample JCL for z/VSE

Following is sample z/VSE JCL. For sample z/OS JCL, see Sample JCL for z/OS (see page 82). For sample C JCL, see Sample JCL for C (see page 74).

Note: Use the following as a guide to prepare your JCL. The JCL statements are for example only. Lowercase letters in a statement indicate a value you must supply. Code all statements to your site and installation standards.

```
 * $$ JOB ...        See the note above and Listing Libraries for CA Datacom Products
(see page 10).
* $$ LST   CLASS=x
// JOB     name
// EXEC      PROC=yourproc  Whether you use PROCs or LIBDEFs, see Listing Libraries
for CA Datacom Products (see page 10).
// OPTION    DECK,NOXREF,DUMP,LOG
// DLBL      WORK1,'precompile.work1',0,SD
// EXTENT    SYSnnn,volser,1,0,reltrk,numtrks
// DLBL      WORK2,'precompile.work2',0,SD
// EXTENT    SYSnnn,volser,1,0,reltrk,numtrks
// DLBL      WORK3,'precompile.work3',0,SD
// EXTENT    SYSnnn,volser,1,0,reltrk,numtrks
```

```
// DLBL      SRCOUT,'source.name',0,SD
// EXTENT    SYSnnn,volser,1,0,reltrk,numtrks
// DLBL      IJSYSIN,'source.name'
// EXTENT    SYSIPT,volser
   ASSGN     SYSLNK,DISK,VOL=volser,SHR
// ASSGN     SYS001,DISK,VOL=volser,SHR
// ASSGN     SYS002,DISK,VOL=volser,SHR
// ASSGN     SYS003,DISK,VOL=volser,SHR
// ASSGN     SYS004,DISK,VOL=volser,SHR
// ASSGN     SYS005,DISK,VOL=volser,SHR
// ASSGN     SYS006,DISK,VOL=volser,SHR
// ASSGN     SYS007,DISK,VOL=volser,SHR
// DLBL      IJSYSLN,'syslnk.dataset',0,sd
// EXTENT    SYSLNK,volser,1,0,reltrk,numtrks
// DLBL      IJSYS01,'ijsys01.work',0,SD
// EXTENT    SYS001,volser,1,0,reltrk,numtrks
// DLBL      IJSYS02,'ijsys02.work',0,SD
// EXTENT    SYS002,volser,1,0,reltrk,numtrks
// DLBL      IJSYS03,'ijsys03.work',0,SD
// EXTENT    SYS003,volser,1,0,reltrk,numtrks
// DLBL      IJSYS04,'ijsys04.work',0,SD
// EXTENT    SYS004,volser,1,0,reltrk,numtrks
// DLBL      IJSYS05,'ijsys05.work',0,SD
// EXTENT    SYS005,volser,1,0,reltrk,numtrks
// DLBL      IJSYS06,'ijsys06.work',0,SD
// EXTENT    SYS006,volser,1,0,reltrk,numtrks
// DLBL      IJSYS07,'ijsys07.work',0,SD
// EXTENT    SYS007,volser,1,0,reltrk,numtrks
* PRECOMPILE
// OPTION    DECK,NOXREF,DUMP,LOG
// EXEC DBXMMPR,SIZE=768K
 CBL LIB,APOST
       cobol source
/*
* COMPILE
// OPTION    NODECK,CATAL
   PHASE   xxxxxxx,*
   INCLUDE DBSBTPR
   INCLUDE DBXHVPR
 ASSGN     SYSIPT,DISK,VOL=volser,SHR
// EXEC IGYCRCTL,SIZE=IGYCRCTL
/*
 CLOSE       SYSIPT,READER
/*
* LNKEDT
   ENTRY   BEGIN
// EXEC LNKEDT
/*
// EXEC
```

```
/*
/&
* $$ EOJ
```

Defining the Procedure to SQL

You can define the procedure to SQL after you have precompiled the program code. This particular procedure would be defined as follows:

```
CREATE PROCEDURE item_order_killer
 (IN canceled_item_id      INTEGER,
  IN distributor-id        INTEGER,
  IN maximum_cancellations INTEGER)
 MODIFIES SQL DATA
 LANGUAGE C (or COBOL or PLI or ASSEMBLER)
 PARAMETER STYLE DATACOM SQL
 EXTERNAL NAME itemkill
```

The procedure name used previously differs from the external name only to illustrate that the external name equates to a load-module name, and the SQL name is an SQL-identifier. For simplicity, we recommend using the same name in both places. Note that procedure parameters may be any valid SQL data type, but the example uses INTEGER to avoid distracting you from the relevant points.

Note also that while OUT and INOUT parameters are supported, they are not used here because this procedure is used with a trigger. Procedures that use output parameters may not be called from a trigger. If there were no need to call the procedure from a trigger and the caller was interested in knowing how many orders had to be canceled, the procedure definition might look something like this:

```
CREATE PROCEDURE item_order_killer
 ( IN    canceled_item_id      INTEGER,
   IN    distributor-id        INTEGER,
   IN    maximum_cancellations INTEGER,
   OUT orders_canceled         INTEGER)
 LANGUAGE C (or COBOL or PLI or ASSEMBLER)
 PARAMETER STYLE DATACOM SQL
 MODIFIES SQL DATA
 EXTERNAL NAME itemkill
```

Note that the CALL PROCEDURE statement must now supply a host variable to receive the "orders_canceled" output parameter from the procedure.

Example: Calling a Procedure

There are two ways to call a procedure. You can:

- Code a CALL or EXECUTE PROCEDURE statement, or
- Create a trigger.

Because the definition of a trigger includes a CALL/EXECUTE statement, in this example a trigger definition is used to illustrate both methods.

Using a Trigger

Given the business process described by the preceding ITEMKILL procedure program, you want to call the procedure every time a row is deleted from the ITEMS_FOR_SALE table. The trigger would look like this:

```
CREATE TRIGGER cancel_orders
  BEFORE DELETE ON sales.items_for_sale
  REFERENCING OLD ROW AS deleted_item
  FOR EACH ROW
  WHEN deleted_item.date_available <= CURRENT_DATE
  CALL item_order_killer(deleted_item.item_id,deleted_item.vendor_id, 5)
```

After the CREATE TRIGGER statement (shown previously) is executed, every DELETE executed against the "items_for_sale" table generates a call (or calls, one per deleted row) to the coded procedure, unless the "date_available" has not been reached, meaning that no orders yet exist.

Using Embedded SQL

When a CALL/EXECUTE PROCEDURE statement is embedded in a program, host variables can be used in the parameter list. Assuming that the procedure in the procedure creation previously shown example included the output parameter "orders-canceled" as the fourth parameter, the CALL statement embedded in the application program might look something like this:

```
CALL item_order_killer(:item_id, :vendor_id, 5, :NUM-ORDERS-CANCELED)
```

The NUM-ORDERS-CANCELED would have to be declared as an integer variable in the program. Note that the first two parameters have also been changed to host variables. This is because outside the CREATE TRIGGER statement, CALL parameters cannot refer to columns in a table. They can, however, be host variables containing column values. Since these are input-only parameters (IN, as opposed to INOUT or OUT), literals and expressions can also be passed. OUT and INOUT parameters must be host variables, since data is returned to the caller.

Left Outer Joins

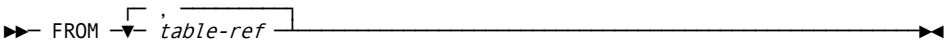
Overview of Joins

A *join* is a query used to return rows that consist of columns selected from more than one table. The combined rows that are returned are selected and *joined* together by each row of each table being evaluated against *join predicates.* A new table therefore results from a join.

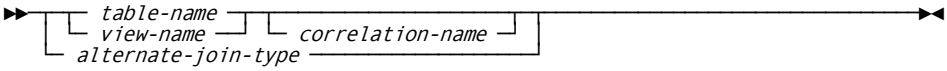
There are different kinds of joins. *Inner joins* eliminate from the resulting table the combined rows that do not satisfy evaluation against the join predicates. Therefore, with inner joins if no matching row is found, no rows are returned. Inner joins were supported by CA Datacom/DB in previous versions and continue to be supported (see Joining Tables). *Outer joins* preserve the rows an inner join would discard by returning those rows with nulls substituted for each column of one of the tables.

The SELECT statement's subselect and select-into syntax uses the FROM clause as an optional choice. As shown in the third of the three following syntax diagrams, JOIN is used in the alternate-join-type segment of the table reference syntax (see the second diagram) in the FROM clause (see the first diagram).

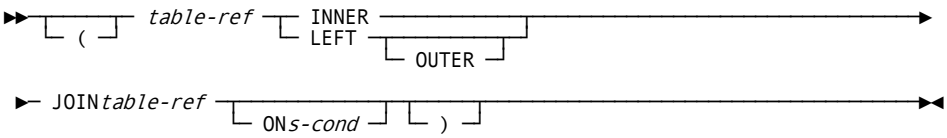
In the following diagrams, while a table reference (table-ref) is shown to be the main component of the FROM clause, it can also be referenced directly from inside the JOIN syntax (third diagram, alternate-join-type syntax).



The table-ref shown in the syntax box immediately preceding the following one has syntax as follows:



The alternate-join-type shown in the syntax box immediately preceding the following one has syntax as follows:

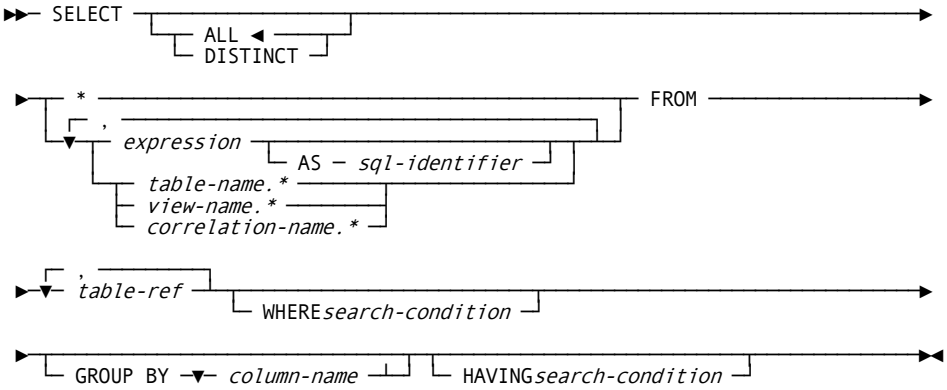


Note: The *s-cond* (search-condition) specified in the optional ON clause differs from the one in the WHERE clause in that the ON clause defines the join conditions that determine which rows contain nulls, as opposed to the WHERE clause, which eliminates rows from the result entirely. Also note that if you use the optional parentheses, they must be balanced. That is, if you use an open parenthesis, you must also use a close parenthesis.

The previously shown JOIN syntax is compatible with Ingres, DB2, and ANSI SQL3 Core SQL.

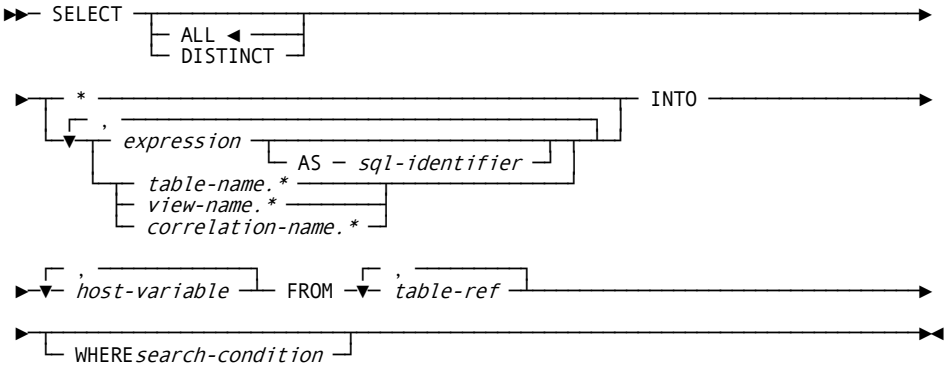
SELECT Statement Subselect Syntax

Following is the SELECT statement's complete subselect syntax with the JOIN syntax as a choice in the FROM clause:



SELECT Statement Select-Into Syntax

Following is the SELECT statement's complete select-into syntax with the JOIN syntax as a choice in the FROM clause:



Inner Join Example

The following example of an inner join *only* returns rows where the CUSTOMER table has a matching row in the ORDERS table:

```
SELECT T1.NAME, SUM(T2.AMOUNT)
FROM CUSTOMER T1, ORDERS T2
WHERE   T1.CUSTNO = T2.CUSTNO
GROUP BY T1.NAME;
```

Outer Join Example

If you want to see *all* customers regardless of whether they have an order, use a LEFT OUTER JOIN in a FROM clause. The keyword LEFT specifies that the table on the left (CUSTOMER in the example) is to be preserved, that is to say, all of the rows in the CUSTOMER table are to survive the join operation. The word OUTER is optional, that is, LEFT JOIN is equivalent to LEFT OUTER JOIN in the syntax. In the following outer join example, a row is returned for each CUSTOMER even if there is no matching ORDERS row.

```
SELECT T1.NAME, VALUE (SUM(T2.AMOUNT), 0)
FROM CUSTOMER T1 LEFT OUTER JOIN ORDERS T2
     ON T1.CUSTNO = T2.CUSTNO
GROUP BY T1.NAME;
```

Using an outer join to see all customers is simpler than the alternate method (shown following) of coding a nested loop in your host application.

```
FOR EACH CUSTOMER
    SET TOTAL = 0
    FOR EACH ORDERS
        WHERE ORDERS.CUSTNO = CUSTOMER.CUSTNO
        SET TOTAL = TOTAL + ORDERS.AMOUNT
    END FOR
    PRINT CUSTOMER.NAME, TOTAL
END FOR
```

Value of Rows That Do Not Match

In the outer join example previously shown, when there is no matching ORDERS row, the *null value* is returned for T2.AMOUNT. This is true even if the AMOUNT column was defined as NOT NULL.

If there is a default value you wish to have returned when a value is null, you can use the VALUE function, which returns the first non-null value in its argument list. In this case, SUM(T2.AMOUNT) is returned if there is a matching row. Zero is returned when:

- There is no matching row, and
- AMOUNT is the null value.

The phrase CUSTOMER T1 LEFT OUTER JOIN ORDERS T2 is called a *joined table.* A joined table can be used with other simple table references in the FROM list. For example, if you want to also report from the LINE_ITEM table:

```
SELECT T1.NAME, VALUE (SUM(T2.AMOUNT), 0), COUNT(DISTINCT T3.ORDNO)
FROM CUSTOMER T1 LEFT OUTER JOIN ORDERS T2
     ON T1.CUSTNO = T2.CUSTNO,
     LINE_ITEM T3
 WHERE T2.ORDNO = T3.ORDNO
 GROUP BY T1.NAME;
```

Alternately you can use the joined table syntax by replacing the right operand with an INNER JOIN:

```
SELECT T1.NAME, VALUE (SUM(T2.AMOUNT), 0), COUNT(DISTINCT T3.ORDNO)
FROM CUSTOMER T1 LEFT JOIN (ORDERS T2 INNER JOIN LINE_ITEM T3
                            ON T2.ORDNO = T3.ORDNO)
     ON T1.CUSTNO = T2.CUSTNO
GROUP BY T1.NAME;
```

In this example, note the following:

- The optional keyword OUTER has been omitted.
- Since both the ORDERS and LINE_ITEM table are on the right-side of the LEFT JOIN, the LINE_ITEM columns returned are also null when there is no matching ORDERS row.
- Since COUNT always returns zero even if its argument, T3.ORDNO, is null, there is no need for the VALUE function.

WHERE Clause

The WHERE clause can be used with the LEFT OUTER JOIN, but since the WHERE clause is conceptually executed after the FROM clause (which includes executing the LEFT OUTER JOIN), references to columns in tables on the right side of a LEFT OUTER JOIN are evaluated against the null value for non-matching rows. This means that unless IS NULL is the predicate (or it is under an OR that has an IS NULL predicate), the predicate result is *unknown,* and the row is eliminated. Eliminating unmatched rows therefore turns your LEFT OUTER JOIN into an INNER JOIN.

For example, if you want to modify the previously shown query to only return ORDERS data for the current date, but you still want to see all CUSTOMER rows:

```
SELECT T1.NAME, VALUE (SUM(T2.AMOUNT), 0), COUNT(DISTINCT T3.ORDNO)
FROM CUSTOMER T1 LEFT JOIN (ORDERS T2 INNER JOIN LINE_ITEM T3
                            ON T2.ORDNO = T3.ORDNO)
     ON T1.CUSTNO = T2.CUSTNO
WHERE T2.ORDER_DATE = CURRENT DATE OR T2.ORDER_DATE IS NULL
GROUP BY T1.NAME;
```

Without the OR T2.ORDER_DATE IS NULL you would not get back CUSTOMERS that have no matching ORDERS rows.

As the following example shows, by placing the predicate in the ON clause you do not have to add the OR IS NULL predicate, because the ON clause is evaluated before the columns of the non-matching row are set to the null value:

```
SELECT T1.NAME, VALUE (SUM(T2.AMOUNT), 0), COUNT(DISTINCT T3.ORDNO)
FROM CUSTOMER T1 LEFT JOIN (ORDERS T2 INNER JOIN LINE_ITEM T3
                            ON T2.ORDNO = T3.ORDNO AND
                               T2.ORDER_DATE = CURRENT DATE)
     ON T1.CUSTNO = T2.CUSTNO
GROUP BY T1.NAME;
```

Performance Considerations

Outer joins are procedural. The SQL Optimizer cannot do reorders of the join sequence without altering the semantics of the join. Therefore, outer joins are executed as written, that is to say depth-first, left-to-right. In the preceding examples, T2 is joined to T3 and the result is joined to T1. The order of predicate evaluation is discussed in more detail on Order of Predicate Evaluation (see page 98)

You should strive to write the joined table in the most efficient order and place predicates in the first join in which they can be evaluated. For example, as previously shown, the restriction on ORDER_DATE could have been added to the outer ON clause, but then it could not be used to limit the previous join, and all ORDERS rows would be joined to all LINE_ITEM rows, only to have many of those rows rejected in the next join step.

As shown in the following example, if you add a restriction on T1.CUSTNO to return the row for a specific customer, this restriction is not applied until all the ORDERS and LINE_ITEM rows have been joined:

```
SELECT T1.NAME, VALUE (T2.AMOUNT, 0), COUNT(DISTINCT T3.ORDNO)
FROM CUSTOMER T1 LEFT JOIN (ORDERS T2 INNER JOIN LINE_ITEM T3
                            ON T2.ORDNO = T3.ORDNO AND
                               T2.ORDER_DATE = CURRENT DATE)
     ON T1.CUSTNO = T2.CUSTNO AND
        T1.CUSTNO = :CUSTNO AND
GROUP BY T1.NAME, T2.AMOUNT;
```

Rather than doing the previous, it is more efficient to write:

```
SELECT T1.NAME, VALUE (T2.AMOUNT, 0), COUNT(DISTINCT T3.ORDNO)
FROM (CUSTOMER T1 LEFT JOIN ORDERS T2
      ON T1.CUSTNO = T2.CUSTNO AND
        T1.CUSTNO = :CUSTNO
         T2.ORDER_DATE = CURRENT DATE) INNER JOIN LINE_ITEM T3
                                      ON T3.ORDNO = T2.ORDNO
GROUP BY T1.NAME, T2.AMOUNT;
```

Order of Predicate Evaluation

Predicates in ON clauses are conceptually evaluated before the WHERE clause. Within the FROM clause, predicates in the ON clauses are evaluated in the order the joins are executed —inner most (deepest) first, and then left to right. For example, in the following FROM clause, T1 is joined to T2, T3 joined to T4, and then the result of T1 and T2 joined to the result of T3 and T4:

```
FROM (T1 left join T2 on t1.c1 = t2.c1)  left join
     (T3 left join T4 on t3.c1 = t4.c1)  on T2.c1 = T4.c1
```

If a non-matching row has caused the columns of the non-preserved row to be set to null in a previous join, then unless IS NULL is the predicate, the result is unknown. In the previously shown example, if either of the first two joins produces a non-matching row, then the T2.c1 = T4.c1 predicate evaluates as unknown, and columns in T3 and T4 are set to the null value.

Since the WHERE clause is evaluated last, a predicate other than IS NULL on a column that has been set to the null value in an outer join causes the result row to be rejected. This effectively changes the outer join to an inner join, since any preserved rows are rejected. Continuing the example, WHERE T4.c2 = 'xxx' effectively converts both left joins to inner joins.

Non-Matching Rows

Conversely, the following query finds only rows of T1 that do *not* have a matching T2 row (when T2.c2 is defined as not nullable):

```
FROM T1 left join T2 on t1.c1 = t2.c1
WHERE T2.c2 IS NULL
```

However, since each matching T2 row is found and then rejected, the following query is more efficient because the NOT EXISTS predicate is evaluated after finding only a single matching row:

```
SELECT *
FROM T1
WHERE NOT EXISTS (SELECT *
                  FROM T2
                  WHERE T2.c1 = T1.c1)
```

Order of Joins

The join order is set when LEFT OUTER or INNER is used, otherwise the SQL optimizer determines join order (unless plan option OPT=M is used). Care must therefore be taken to ensure efficient evaluation. (This means INNER JOIN can be used in place of OPT=M to manually specify join order at the query level.)

LEFT joins always use the *nested loop* join method.

NULL Indicator Variables

When selecting a NOT NULL column that could be set to the null value by an outer join, you must supply a host indicator variable for the column.

SQL Memory Guard

The SQL Memory Guard function, used for SQL memory monitoring, is not required for normal production processing, but if you are experiencing an unusual SQL memory problem that is affecting the availability of the Multi-User Facility, the SQL Memory Guard function can be invoked. You should in most cases, however, only use the SQL Memory Guard function if CA Support requests you to do so.

SQL Memory Guard monitors memory requests made by SQL and looks for possible misuse and contention on each memory address as well as the entire memory pool. If you invoke the SQL Memory Guard function, you may notice a slight increase in CPU usage, because the memory guard monitors and records every SQL memory request.

When the invoked SQL Memory Guard encounters a possible memory problem, it aborts the invalid memory request before any damage is done to the memory pool. The request that generated the invalid memory request then receives an SQL return code of -999, thus protecting the Multi-User Facility from a possible destructive memory failure.

The following error message is produced when problems are detected. Additional debugging information is dumped to the Statistics and Diagnostics Area (PXX).

-999

INTERNAL ERROR (file-name LINE xyz): command CONFLICT - addr1(task1)/addr2(task2)

If such a -999 message is received, contact CA Support and give them the following information from the message text:

command

Is the service requested by task1, such as FREEMEM or DELPOOL.

addr1

Is the address of the illegal call to memory services (the R14), relative to the entry point of load module DBSRPPR.

task1

Is the task (RWTSA) number of the *outlaw* requestor of memory services.

addr2

Is the relative address of the call for the prior conflicting request (valid regardless of RWTSA contents).

task2

Is the task (RWTSA) number of the caller for the prior conflicting request on either the memory address or the memory pool in question. Note that this task may be executing an unrelated job by this point in time. This does not, however, necessarily prevent the SQL Memory Guard from finding the problem.

The above information helps CA Support resolve memory-related problems more quickly. In addition, as always when you contact CA Support to report any SQL error code, provide a Statistics and Diagnostics Area (PXX) report with DUMPS=FULL.

Following is an example of an actual -999 message:

-999

INTERNAL ERROR (MEMSERV LINE 1396): FREEMEM CONFLICT - FD14(1)/FCBE(1)

Activating the SQL Memory Guard

The features of the SQL Memory Guard can be activated or deactivated, though normally only at the direction of CA Support, by executing the following commands (for more information on DIAGOPTION, see the *CA Datacom/DB Database and System Administration Guide*):

DIAGOPTION 2,2,ON

Causes automatic debugging and abend prevention to begin either immediately or as soon as the trace table is allocated.

DIAGOPTION 2,2,OFF

Causes automatic debugging and abend prevention to stop. If the trace table has been allocated, requests are still logged.

DIAGOPTION 2,4,ON

Causes the memory trace table to be allocated (approximately 32K) and activated (meaning memory requests are logged) when the next new RUN UNIT starts its first SQL request. In order to receive automatic debugging and abend prevention, also execute DIAGOPTION 2,2,ON as described in the following related section.

DIAGOPTION 2,4,OFF

Causes the memory trace table to be freed (the memory manager makes the storage available for use by SQL only) when the next new RUN UNIT starts its first SQL request.

SQL and Multiple Multi-User Facilities Support

Beginning in r11, there is support for multiple Multi-User Facilities.

With SQL programs, one compile unit must run against one CA Datacom Datadictionary, have one plan, and execute against the Multi-User Facility containing both. An SQL program may execute with a multiple Multi-User Facilities User Requirements Table, but all SQL requests execute using the *first* Multi-User Facility in the connection list. In this case it would have the ability to specify a SIDNAME= and not be forced to use DBSIDPR. An SQL program could be split into multiple modules and each could connect to a different Multi-User Facility, but no joining of tables or constraints outside of one Multi-User Facility are allowed.

SQL detects RRS User Requirements Tables, issues a return code, and does not process COMMIT or ROLLBACK requests. The multiple Multi-User Facility interface detects the specific errors and generates RRS COMMIT or ROLLBACK calls. If RRS is successful, the SQLCODE and related error fields are set to zero/blank. If RRS provides a return code on the SRRCMIT/SRRBACK, the SQL error is changed to:

- -117,
- program DBMMIPR, and
- message:

  RETURN CODE <94(121)> ON COMMAND <COMIT> R15 <?>

  where ? is the R15 error value from RRS.

Note: For more information on multiple Multi-User Facilities, see the *CA Datacom/DB Database and System Administration Guide*.

Application Design Considerations

Be aware of the application design considerations involving index-only processing and cursor processing.

Index-Only Processing

In index-only processing data is retrieved, when possible, from the index only, eliminating the cost of accessing the actual row in the data area. For example, consider an online application to look up an account by name when the customer does not know their account number:

```
SELECT NAME, STREET, CITY, ST, ZIP, ACCOUNT_NBR
FROM ACCOUNTS
WHERE NAME BETWEEN :NAME_BEG AND :NAME_END
OPTIMIZE FOR 20 ROWS;         -- ONLY 20 ROWS ON A SCREEN
```

This query can use index-only processing if the following two conditions are met:

- There is a key with all the columns in the query. This includes columns in the SELECT list, WHERE clause, and anywhere else in the query. In this case key (NAME, ST, ZIP, CITY, STREET, ACCOUNT_NBR) would work. Notice that the order of the columns in the key is not important, except for being good for selection. Therefore, the NAME column is first and the remaining columns are included only so that all the columns referenced are available from the index.
- Isolation level U is specified for the plan, so that no lock is required.

If the ACCOUNTS table Native Key is ACCOUNT_NBR, accessing rows in NAME sequence could cost an I/O for every row retrieved. Indexes usually have 100 or more entries per DXX block, requiring just one logical I/O instead of 21 I/Os.

Note: If you add columns to an index for index-only retrieval that are updated frequently, the cost of updating the index may outweigh the advantage of index-only retrieval.

Cursor Processing

The result set of rows for a cursor may be a static set of rows copied to a temporary table before any rows are returned to your program, or the rows may be retrieved dynamically as you FETCH the rows. You may use isolation levels C or R to isolate your transactions from other concurrent transactions, but this does not isolate your cursor from changes your transaction is making while you are FETCHing rows from a cursor that is dynamically retrieving rows. For example, if you update a row that has been FETCHed with a separate searched UPDATE statement, the same row may be returned in a subsequent FETCH if the updated values place it in the result set ahead of the dynamic retrieval process.

Note: If you use an UPDATE where current of <-cursorName>, CA Datacom/DB insures that the row is not returned again (at the expense of building a temporary index of updated URIs).

To isolate your cursor from changes made by other statements in your program while the cursor is open, you can use an ORDER BY, that cannot be satisfied by any key, to force a temporary table to be built. However, if such a key is added in the future, this new key may be used to eliminate building a temporary table. Also, even if you specified isolation level C, the current row of the cursor may not be locked, because a block of rows is returned to your program and the cursor stability lock on the row has already been released.

To ensure the current row of the cursor is locked, you can specify an UPDATE or DELETE where current of <-cursorName> for the cursor. Even if you never execute these statements, blocking is not used, and the current row of the cursor is held by an exclusive lock until you FETCH the next row.

DBSQLPR

DBSQLPR is a utility that enables users to execute SQL statements through CA Datacom/DB. It provides a superset of the functionality previously provided by the SQIDEMO and COB430 utilities without any dependence upon the SQI interface or any requirement to preprocess and compile an SQL program. DBSQLPR is for users who want quick access to SQL functionality but do not have access to CICS or the CA Datacom Server.

Consider the following:

- If you have existing SQIDEMO and COB430 JCL, it runs with little or no changes.
- SQL statements must be terminated with a semicolon (;) unless the TERM= parameter is used (see DBSQLPR Options (see page 105)).
- SQL statements may not contain host variable references.
- All nullable columns are printed with one extra character to the left. This character is blank when a value is present and an asterisk (*) when the data is NULL.
- The CA Common Services for z/OS CAILIB is required in STEPLIB (for z/OS) or LIIBDEF (for z/VSE).
- When using DBSQLPR, specify PLNCLOSE=T or allow PLNCLOSE to default to that value.

Processing

The execution flow for DBSQLPR is as follows:

1. DBSQLPR examines any input parameters passed programmatically or through the command line (the PARM= specification in the JCL). See DBSQLPR Options (see page 105).
2. The options file is opened and processed. With the exception of the OPTFILE= parameter, which may *only* appear on the PARM= input parameter string, any option may appear in either the options file or the PARM= specification. Those specified on the PARM= input parameter string override any duplicate specification in the options file.

   Note: The same set of plan options you can specify in the COBOL Preprocessor can be specified in DBSQLPR. See Specifying Processing Options in COBOL.
3. The SYSIN file is read, and the input is processed as a series of commands (see Line Commands (see page 104)) and SQL statements.

Line Commands

The following in-line commands are accepted by DBSQLPR in your input file:

DROP PLAN *auth-id.plan-name*

You can submit a DROP PLAN statement in Version 12 and above as an in-line command in DBSQLPR by using DROP PLAN *auth-id.plan-name* (where *auth-id* is an optional authorization ID, followed by a period, and the *plan-name* is the required name of the plan you want to drop).

We recommend that you do not use DROP PLAN to drop plans related to procedures. and functions. Use the DROP PROCEDURE statement and DROP FUNCTION statements to drop procedures.

For more information about DROP PLAN, see DROP PLAN (DBSQLPR) (see page 114).

--

Two dashes at the start of any line causes the line to be interpreted as a comment and ignored.

*****

An asterisk at the start of any line causes the line to be interpreted as a comment and ignored, unless that line contains one of the following line commands that begin with an asterisk:

***$COLUMN (or *$C)**

Specifies one data item per line of output (same as *$THIN).

***$PAGE (or *$P)**

Generates a form-feed and page-header.

***$ROW (or *$R)**

Specifies tabular output with column headings truncated to width of data.

***$THIN (or *$T)**

Specifies one data item per line of output (same as *$COLUMN).

***$WIDE (or *$W)**

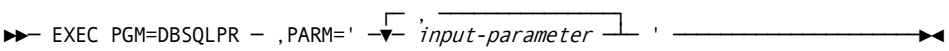
Prints output in tabular form, reverting to *$THIN if data exceeds PRTWIDTH= specification.

***$ZERO (or *$Z)**

Zeroes the job-step return code.

DBSQLPR Syntax

Following is the syntax for DBSQLPR:



Note: In z/VSE environments, parameter separators in the PARM= input parameter string must be spaces.

DBSQLPR Options

DBSQLPR examines any input parameters (shown as *input-parameter* in the syntax diagram) that are passed through the OPTIONS file or the command line (the PARM= specification in the JCL). Any option can appear in either the OPTIONS file or the PARM= specification with the exception of OPTFILE=, which can only appear in the PARM= specification.

Note: In addition to the following, all plan options valid for the COBOL Preprocessor are also valid. See Specifying Processing Options in COBOL.

AUTHID=

Determines the default authorization ID for non-qualified SQL names.

Valid Entries:

an authorization ID name of from 1 to 18 characters

Default Value:

SYSADM

DATASEPARATOR=*c*

DATASEPARATOR produces output in a form ready for import into spreadsheet software. The separator character you specify is placed after each data item returned from a SELECT statement.

The *c* specifies the data-separator character and is generally a comma but can be most non-blank characters that work for your data. We recommend that you do a test with your choice of separator character to determine whether it works as desired.

This option works only for data that can be represented in tabular format. The combination of SQUISH and a large PRTWIDTH specification (up to 1500 is allowed) can be used to force some non-tabular output to become tabular.

Specification of a blank is not allowed because of the way z/VSE handles execution parameters, but blank-delimited output is easily produced by either of the two following methods:

- Specify DATASEPARATOR=B, where B is interpreted as a blank.
- A space is automatically employed as the separator when the DATASEPARATOR= option is not used. In this case, you could get two spaces in a row when null indicators indicate non-null, or you could get multiple spaces if SQUISH is not specified. Specifying SQUISH compresses the data and eliminates the solid line that appears underneath the column headers, because the columns are no longer fixed-length when using SQUISH.

For data-only output (column-names, data types, and data only), specify NOECHO and NOPAGES with DATASEPARATOR=. If you want column-names and data types eliminated, add NOCOLHDR. NOTYPE can be used instead of NOCOLHDR if you do not want the data types but still want to see the column-names. If you need to eliminate unneeded spaces, add SQUISH. Any column containing a null-indicator still has a space or asterisk preceding the data-value.

Valid Entries:

a comma, or any non-blank character that works for your data,
or a B (for a blank space)

Default Value:

a blank space

ERRABORT=

Specifies that a certain SQLCODE, if encountered, aborts the execution of DBSQLPR.

Note: The in-line command *$ZERO zeros the jobstep return code.

Valid Entries:

any valid SQLCODE, for example, ERRABORT=-117

Default Value:

If not specified, this feature is inactive.

ERRMIN=

Specifies the minimum SQLCODE that does not abort DBSQLPR. That is, the format of the specification is ERRMIN=*sqlcode,* where *sqlcode* is the lowest numbered SQLCODE that does not cause DBSQLPR to abort. For example, if you wanted DBSQLPR to terminate on any negative SQLCODE, you would code ERRMIN=0. If you wanted even positive (warning) SQLCODEs to abort the program, you could code 9999.

Note: The in-line command *$ZERO zeros the jobstep return code.

Valid Entries:

-9999 through 9999

Default Value:

-9999
The -9999 default means that DBSQLPR does *not* terminate on any SQLCODE.

FORMFEED=*character*

FORMFEED= changes the FORM-FEED character. Also see NOFORMFEED.

Valid Entries:

any decimal value between 1 and 255 that works correctly in your environment

Default Value:

12 decimal (z/OS) or 241 decimal (z/VSE)

HEXCHAR

Specify HEXCHAR to request hexadecimal output for all CHAR data. If you do not specify HEXCHAR, you get character output with binary zeros and new-lines blanked, and all other control characters printed.

HEXGRAPHIC

Specify HEXGRAPHIC to request hexadecimal output for all GRAPHIC data. If you do not specify HEXGRAPHIC, you get character output.

INFILE=

Specify INFILE= to request an alternate DDNAME (in z/OS) or DTFNAME (in z/VSE) for the STDIN/SYSIN input.

Note: Specification of STDIN (the default) or any other file automatically opened by the C runtime environment causes a *duplicate open* error.

Valid Entries:

a valid alternate DDNAME (in z/OS) or DTFNAME (in z/VSE)

Default Value:

STDIN

INPUTWIDTH=

Specifies a column beyond which the specified SYSIN lines are to be ignored. May be used to ignore line numbers or other unwanted information to the right of your intended input line but preceding any line-break.

Valid Entries:

50 thru the maximum your system supports, up to 99999

Default Value:

999

NOCOLHDR

NOCOLHDR eliminates column headers from tabular output. Form-feeds and page headers still print, unless NOPAGES is also specified.

NOECHO

Specifying NOECHO indicates that only the data and any error summaries are printed. If you do not specify NOECHO, user input is echo-printed, that is, not only the data and error summaries are printed.

NOFORMFEED

NOFORMFEED works the same as NOPAGES. Both suppress form-feeds and CA Datacom copyright page headers. Column-name headers for tabular output still occur exactly once at the top of the output, unless NOCOLHDR is also specified. NOPAGEHDR also suppresses CA Datacom copyright page headers but does not suppress form-feeds.

NOPAGEHDR

Specify NOPAGEHDR to suppress page headers. If you do not specify NOPAGEHDR, page headers are therefore not suppressed.

NOPAGES

NOPAGES works the same as NOFORMFEED. Both suppress form-feeds and CA Datacom copyright page headers. Column-name headers for tabular output still occur exactly once at the top of the output, unless NOCOLHDR is also specified. NOPAGEHDR also suppresses CA Datacom copyright page headers but does not suppress form-feeds.

NOTYPE

Specify NOTYPE to request that data types be omitted from the printed output. If you do not specify NOTYPE, data types are not omitted from the printed output.

OPTFILE=

Specifies an alternate DDNAME (in z/OS) or DTFNAME (in z/VSE) for the options file, but OPTFILE= itself can only be specified in the PARM= specification.

Valid Entries:

a valid DDNAME (in z/OS) or DTFNAME (in z/VSE)

Default Value:

OPTIONS

PAGELEN=

This parameter specifies the number of output lines per page. Use a high number if you do not like the page headers.

Valid Entries:

40 through 2147483647

Default Value:

the page length specification in the LINES= parameter in your DBSIDPR module, or 56 if DBSIDPR contains a number lower than LINES=40 (see the *CA Datacom/DB Database and System Administration Guide* for more information about DBSIDPR)

PLANAME=

Specifies the name of the plan to create for this execution.

Valid Entries:

a valid plan name

Default Value:

DBSQL*x,* where *x* consists of selected portions of the system clock

PLANNAME=

Specifies the name of the plan to create for this execution. It is compatible syntax for PLANAME=.

Valid Entries:

a valid plan name

Default Value:

DBSQL*x,* where *x* consists of selected portions of the system clock

PRTFORMAT=

Use this parameter to specify tabular or column output. The default is WIDE (or W), to print wide (tabular output) when possible.

ROW (or R) gives tabular output but specifies that the width of the print is only as wide as the data, truncating the column name and data-type-descriptor-string to the length of the data, even if the data is only one byte long, which allows more data to fit onto each line.

THIN (or T) and COLUMN (or C) print one column value per line.

Note: These functionally equivalent in-line commands can be used: $WIDE (or *$W), *$ROW (or *$R), *$THIN (or *$T), *$COLUMN (or *$C). In-line commands allow the format to be changed *on the fly*.

Valid Entries:

WIDE or W, ROW or R, COLUMN or C, THIN or T

Default Value:

WIDE

PRTFILE=

This parameter specifies an alternate DDNAME (in z/OS) or DTFNAME (in z/VSE) for the STDOUT output.

Valid Entries:

a valid alternate DDNAME (in z/OS) or DTFNAME (in z/VSE)

Default Value:

STDOUT

PRTMODE=

Do not use this parameter unless directed to do so by CA Support.

PRTWIDTH=

Maximum row width for PRTFORMAT=ROW rows. Specifies where the line is to be split. That is, you can use PRTWIDTH= to define the width that can be printed before either data truncation or a forced switch from tabular (wide) to column-at-a-time (thin) output occurs. However, be aware that some types of output files wrap lines at the specification for LRECL=, while some output file types truncate. PRTWIDTH= should therefore be used to tell DBSQLPR when to force column-based output to occur, unless for some reason you want file type-dependent behavior to occur. Also be aware that, to prevent column values from spanning line boundaries, the line may be split earlier than specified by PRTWIDTH=.

Valid Entries:

80 through 1500

Default Value:

132 (tabular output, when possible, is the default)

ROWLIMIT=

This parameter truncates FETCH sequences that retrieve too many rows. That is, you can use ROWLIMIT= to truncate FETCH sequences that retrieve more rows than you want to retrieve.

Note: If the ROWLIMIT is exceeded, DBSQLPR returns a -2009 DBSQLPR error code and issues an error message. If you know a particular query may exceed the limit and still want a job-step return code of 0, place the *$ZERO line command following that query in your input file.

Valid Entries:

0 through 999999999

Default Value:

1000

SQUISH

SQUISH removes unneeded spaces from column headers and data that can then, if enough, be output in tabular format. Do not use SQUISH for non-tabular data. SQUISH can, when used with a large PRTWIDTH, enable non-tabular output to become tabular, but SQUISH does not eliminate spaces that have been generated to represent null indicators.

SQUISH can cause column data output to vary in length. If this creates a problem for you, try using PRTFORMAT=R (ROW) or line command *$R (*$ROW) as an alternative method to reduce column widths while preserving the tabular appearance of some output that is useful for certain features of spreadsheet packages.

TBLHDRRPT=*rows*

Specifies how frequently to repeat the header lines that precede table-format output. Specify this parameter only if you *do not* want report-headers at the top of each page of a query's output, but prefer instead to see them at longer or shorter intervals. If the number you specify matches your PAGELEN= specification or default, adjustments are made to help ensure that table-headers appear at the top of each new page, even if NOPAGEHDR has been specified. Otherwise, this is the number of rows printed before subsequent table-headers appear.

Note: You can use the *$PAGE line command to help ensure that your output starts at the top of a page. Specifying the *$PAGE line command before a query helps ensure a full page of output before a page-break and a new set of table-headers appears.

Valid Entries:

40 through 2147483647
This range of valid values is the same as the valid values for PAGELEN=.

Default Value:

Produces one set of headers on each new page (even if NOPAGEHDR is specified, as previously described).

TERM=

Specifies a character to terminate SQL statements.

The terminating character is changed to a semicolon (;) in the SQL statement that is passed to the DBMS and therefore, regardless of what terminating character you specify with the TERM= parameter, appears in DBSQLPR output reports as a semicolon.

SQL statements that appear inside the compound statements of SQL Procedures are terminated by semicolons, but semicolons are also used as the default termination character in DBSQLPR for complete SQL statements. The DBSQLPR parameter TERM= can be used, however, to prevent DBSQLPR from truncating compound statements in a CREATE PROCEDURE statement at the first semicolon. We therefore recommend that you add TERM=@ (specifying an @ symbol) to your DBSQLPR command line or options-file options. Then, although semicolons are still used inside the compound statements embedded in your CREATE PROCEDURE statement, at the end of each complete SQL statement, including the CREATE PROCEDURE statement itself, the @ symbol can be used as the termination character instead of a semicolon to avoid this ambiguity.

Valid Entries:

May be any character that is not alphanumeric and not valid as part of an SQL statement. Valid SQL-statement characters include, but are not limited to SQL-identifier characters, parentheses, dollar signs, percent signs, underscores, commas, quotes, apostrophes, asterisks, pound signs, and arithmetic and bitwise operators. One example of a valid terminating character is the *at* sign (@).

Default Value:

a semicolon (;)

TRACEALL

The inclusion of this keyword causes all traces internal to DBSQLPR to be printed.

Important! Use this option only when CA Support tells you to use it and only as CA Support instructs you to use it.

TRACEDETAIL

The inclusion of this keyword causes certain traces internal to DBSQLPR to be printed. Traces specifically related to calls to CA Datacom are printed.

Important! Use this option only when CA Support tells you to use it and only as CA Support instructs you to use it.

TRACERAAT

The inclusion of this keyword causes certain traces internal to DBSQLPR to be printed. Traces specifically related to calls to CA Datacom are printed.

Important! Use this option only when CA Support tells you to use it and only as CA Support instructs you to use it.

DROP PLAN (DBSQLPR)

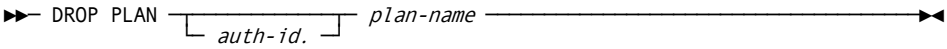
(Executable only from DBSQLPR.)

A plan is the representation in SQL of an SQL application and, unless removed with the DROP PLAN statement, can continue to occupy table space long after the application has been retired. When you drop a plan, make certain that the plan is not related to an SQL application that is still in use.

We recommend that you do not use DROP PLAN to drop plans related to procedures. and functions. Use the DROP PROCEDURE statement and DROP FUNCTION statements to drop procedures. and functions.

The syntax for the DROP PLAN statement is as follows.

Note: The DROP PLAN statement is executable only from DBSQLPR, but it can also be submitted as a DBSQLPR in-line command (see Line Commands (see page 104)).



auth-id.

(Optional) The *auth-id.* is the authorization ID, followed by a period, related to the plan name that you want to drop.

Valid Entries:

a valid authorization ID

Default Value:

(No default)

plan-name.

The *plan-name* is the name of the plan that you want to drop.

Valid Entries:

a valid plan name

Default Value:

(No default)

Example JCL

Note: Users of z/OS can use either spaces or commas as parameter separators in the PARM= input parameter string in the JCL. In z/VSE environments, the SYSLST file must be assigned to a POWER-controlled print device. Parameter separators in the PARM= input parameter string must be spaces.

The following example shows the PARM= input parameter string specification.

```
PARM='PRTWIDTH=255,INPUTWIDTH=72,PAGELEN=56,TBLHDRRPT=56'
```

IBM limits the PARM= input parameter string to 100 bytes. The following example depicts lines that have been spanned. Be aware the first line ends in column 71 and the second line starts in column 16.

```
PARM='PRTWIDTH=255,INPUTWIDTH=72,ERRBORT=-56,OPTFILE=OP,ROWLI
      MIT=9'
```

z/OS JCL Sample

Note: Use the following sample as a guide to prepare your JCL. The JCL statements are for example only. The lowercase letters in a statement indicate a value that you must supply. Code all statements to your site and installation standards.

```
//jobname      See the note above.
//             CLASS=K,MSGCLASS=X,REGION=1024K
//SQLEXEC  EXEC PGM=DBSQLPR,...see Listing Libraries for CA Datacom Products (see
page 10)
//*
//       PARM='prtWidth=999,inputWidth=80'
//*
//SYSUDUMP DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//STDERR   DD  SYSOUT=*
//STDOUT   DD  SYSOUT=*
//OPTIONS  DD  *
AUTHID=SYSUSR
/*
//SYSIN    DD  *
create table testTable (colChar char(18), colInt integer);
insert into  testTable values ('colChar row 1', 1);
insert into  testTable values ('colChar row 2', 2);
-- Output appears as a table unless "PRTWIDTH=" is exceeded.
select colChar, colInt from testTable;
rollback work;
/*
```

z/VSE JCL Sample

Note: Use the following sample as a guide to prepare your JCL. The JCL statements are for example only. The lowercase letters in a statement indicate a value that you must supply. Code all statements to your site and installation standards.

```
* $$ JOB ...           See the note above..
* $$ LST CLASS=x
// JOB     jobname
*           CREATE OPTIONS FILE USING DITTO
// UPSI 1
// EXEC    PROC=yourproc
// ASSGN   SYSnnn,DISK,VOL=volser,SHR
// DLBL    OPTIONS,'dataset.name',0,SD
// EXTENT  SYSnnn,volser,1,0,reltrk,1
// EXEC DITTO
$$DITTO CSQ FILEOUT=OPTIONS,CISIZE=512,BLKFACTOR=1
AUTHID=SYSADM
/*
$$DITTO EOJ
/*
*           EXECUTE DBSQLPR
// EXEC DBSQLPR
   SELECT * FROM table;
/*
/&
* $$ EOJ
```

Sample Report

```
Date: mm/dd/ccyy    ********************************************************************************        Page:     1
                    *                          CA Datacom/DB SQL Option                            *
Time: hh.mm.ss      *                            DBSQLPR SQL Processor                             *      Version: 12
                    *                  Copyright © 2009 CA. All rights reserved.                   *              SPnn
                    ********************************************************************************


 Command Line Options
 ____________________
 INPUTWIDTH=72
 OPTFILE=OP
 PRTWIDTH=999


 Option File Options
 ____________________
 AUTHID=SYSADM

 INPUT STATEMENT:
 create table testTable (colChar char(18), colInt integer);

 ___ SQLCODE=0, SQLSTATE=00000 ___


 INPUT STATEMENT:
 insert into  testTable values ('colChar row 1', 1);

 ___ SQLCODE=0, SQLSTATE=00000, ROWS AFFECTED=1 ___


 INPUT STATEMENT:
 insert into  testTable values ('colChar row 2', 2);

 ___ SQLCODE=0, SQLSTATE=00000, ROWS AFFECTED=1 ___


 -- Output appears as a table unless "PRTWIDTH=" is exceeded.
 INPUT STATEMENT:
 select colChar, colInt from testTable;

 COLCHAR             COLINT
 CHAR(18)            INT
 ___________________ ___________
  colChar row 1                1
  colChar row 2                2
 ___ 2 rows returned ___


 INPUT STATEMENT:
 rollback work;

 ___ SQLCODE=0, SQLSTATE=00000 ___
```

```
Date: mm/dd/ccyy    ********************************************************************************        Page:     1
                    *                          CA Datacom/DB SQL Option                            *
Time: hh.mm.ss      *                            DBSQLPR SQL Processor                             *      Version: 12
                    *                  Copyright © 2009 CA. All rights reserved.                   *              SPnn
                    ********************************************************************************

 ================================================
 == DBSQLPR is completing with return code 0000 ==
 ==                                             ==
 ==         Statements Found:   00005           ==
 ==         Statement Errors:   00000           ==
 ==         Statement Warnings: 00000           ==
 ================================================
```

DATACOM VIEWs

Overview

This section outlines support for SQL access to legacy data that does not conform to the standard relational model. Access to the following is now supported:

- The elements of an array, that is, single dimensional arrays that correspond to "simple repeating fields" in CA Datacom Datadictionary, are supported.
- "Redefines," that is, columns that redefine other columns (including arrays).

Compound fields are accessible (as CHAR) even if the underlying simple fields are also visible. To achieve this, DATACOM VIEW, a CA Datacom/DB extension to the CREATE VIEW statement (see CREATE VIEW), has been implemented. A DATACOM VIEW differs from a standard view as follows:

- The syntax starts with CREATE DATACOM VIEW instead of CREATE VIEW.
- Columns representing the "redefines" and the array elements that we support are now visible, but only to the CREATE DATACOM VIEW statement. The data is accessed through the view.

CREATE DATACOM VIEW is a very flexible, powerful construct. References to columns in a redefinition from a query or view that references the DATACOM VIEW are references to the columns of the view that are projected by the view. These view columns can inherit the column names of the database table, unless that column is an array, or be assigned new names. The user of the view can consider the view for each record type as a separate database table.

Redefinitions

A redefinition is a column definition or a set of column definitions that provides an alternate definition for another column. Columns in a redefinition do not add length to a row but redefine columns previously listed in the table. Columns of a redefinition may have a different data type.

Redefinitions are commonly used to describe a table with multiple record types. Leading columns are common to all record types, but trailing columns depend upon the record type that is specified by one or more of the leading common columns. Reference to columns dependent upon the record type must be delayed until predicate(s) that reference the record type column(s) have been evaluated, or attempts to reference data contained in rows of the wrong record type could occur.

To prevent these invalid references, the following rules are required:

- Columns contained in a redefinition are visible only to the CREATE DATACOM VIEW statement. CA Datacom/DB enforces this rule.
- Any predicates required to limit the rows of the view to a certain "record type" or otherwise prevent access to invalidly typed data must appear as the first predicates in the WHERE clause and be RQA-able, where "RQA-able" means that the predicates should be simple and presentable in a RQA (Request Qualification Area) used by Compound Boolean Selection (CBS) logic. For example, LIKE predicates are generally not RQA-able, but predicates that use the basic comparison operators are. For more information on Compound Boolean Selection in CA Datacom/DB, see the *CA Datacom/DB Programming Guide*. Failure to follow this rule could result in an attempt to process non-data-type-conformant data, resulting in "invalid data" errors and various other problems. CA Datacom/DB *does not* enforce this rule. *The user must enforce this rule,* that is to say, because CA Datacom/DB has no way of knowing how users distinguish which rows belong to what record type or whether there are separate record types at all, users must supply predicates in the CREATE DATACOM VIEW as needed to prevent CA Datacom/DB from retrieving rows of, for example, "record type B" when reading from the "record type A" view.

  Note: If the data does not contain alternate record types, no predicates to distinguish between record types are needed.

Following are additional rules:

- The FROM clause of the DATACOM VIEW may only reference a single database table. A DATACOM VIEW can therefore not be nested. However, queries and standard views may join DATACOM VIEWS, and those views may be nested.
- See Additional Items to Consider (see page 121) for additional rules that apply to any arrays referenced by the view.

Example of Multiple Record Types

In the following example, the ORDERS table contains orders for both parts and service calls. The table has the following set of column definitions:

- order_id is INTEGER
- order_type is CHAR(7)
- order_detail is CHAR(200)

The order_type column defines the type of row. When order_type is PARTS, order_detail is an array of part numbers, followed by an array of quantities. When the order_type is SERVICE, order_detail contains a textual description of the requested service.

Assume that order_detail is redefined for PARTS orders as follows:

- Part_number is an array of zoned-decimal (NUMERIC(5,0)) part numbers, followed by quantity, an array of INTEGER quantities for each part ordered.
- Order_detail is again redefined for SERVICE orders as follows: service_description is CHAR(199), followed by on_site, which is CHAR(1).

Given the previously stated conditions, the following views can be defined:

```
CREATE DATACOM VIEW part_orders
       (order_id, order_type, part_1, quantity_1,
        part_2, quantity_2, part_3, quantity_3)
 AS
 SELECT order_id, order_type, part_number[1], quantity[1]
        part_number[2], quantity[2], part_number[3],quantity[3]
 FROM orders
 WHERE order_type = 'PARTS'
CREATE DATACOM VIEW service_orders (order_id, order_type, on_site, description)
 AS
 SELECT order_id, order_type, on_site, service_description
 FROM orders
 WHERE order_type = 'SERVICE'
```

The views may then be manipulated almost as if they were database tables, with the restrictions previously noted as well as restrictions applicable to any view. If we had omitted the WHERE clauses, then view part_orders might try to read the service description of a service order and interpret it as an array of zoned-decimal part numbers.

Note: The WITH CHECK OPTION is supported.

Arrays

Array element references are made using the syntax column-name[subscript], as shown in the following example. Non-subscripted array references are not allowed. Array element references must be assigned specific column names in the view, using either the AS-clause in the view-defining query, or an explicit view-column-name list. Outside of the CREATE DATACOM VIEW statement, references to these columns are made using the column names of the view.

Following is an example of a DATACOM VIEW containing references to array elements. The *sales* column does not have to be a redefinition to be visible to CREATE DATACOM VIEW:

```
CREATE DATACOM VIEW (sales01, sales02, sales03, sales04,
        sales05, sales06, sales07, sales08,
        sales09, sales10, sales11, sales12) AS

SELECT  sales[1], sales[2], sales[3], sales[4], sales[5], sales[6],
        sales[7], sales[8], sales[9], sales[10], sales[11], sales[12]

FROM <-tableName>;
```

As shown in this example, elements of an array may be referenced with a literal integer subscript within square brackets within the range of 1 to the number of occurrences. For example, to reference sales for October: *sales[10].*

As already mentioned, reference to an element of an array is restricted to DATACOM VIEWs.

Additional Items to Consider

Consider the following additional points:

- Multiple dimensions are not supported. In this case, arrays within the first dimension are referenced as a single CHAR column as they always have been.
- An array must be defined with CA Datacom Datadictionary, versus CREATE TABLE.
- View columns containing array elements may not be referenced by left-outer joins. Be aware that this reference can be rejected at any point prior to the execution of the query.

- Reference to the array column without the use of subscripts is disallowed.

Note: If a whole-array is referenced in a SELECT list outside a DATACOM VIEW, it is returned as a single CHAR column. The host application may redefine this area as an array. But when using Dynamic SQL, the DESCRIBE statement or the SQLDA of a FETCH statement cannot describe a column as an array (SQLDA is the SQL descriptor area used to describe columns passed to/from the user's application and the Multi-User Facility). This is because the SQLVAR structure of the SQLDA as defined by ANSI/ISO SQL does not include a field for number of elements (SQLVAR is a structure that defines a single column in an SQLDA).

Default Values for Redefinitions and Arrays

As with any INSERT, when performing an INSERT into a DATACOM VIEW, every base-table column must receive an explicitly-specified value, a default, or a NULL. SQL processes any default values defined for a redefined column.

Note: d The default value for an array column, when referenced through a DATACOM VIEW, is considered to be an array-element value and is applied separately to each array element not receiving a value.

These defaults are added through CA Datacom Datadictionary rather than through SQL DDL.

With redefines involved, SQL must decide which default to apply to a given base-table column. Therefore, when processing an INSERT, the following algorithm is applied (columns that are not part of any redefinition are referred to as *primary columns*):

1. The INSERT statement is compared to the definition of the base-table in order to determine whether a value has been supplied for every primary column. If every primary column has a value, the INSERT procedure goes forward. Otherwise, step 2 occurs.
2. For each primary column not receiving a value, CA Datacom/DB:
    a. Lays down any non-NULL default defined for that column.
    b. Overlays this with defaults or NULL for all redefinitions that are both a part of the DATACOM VIEW, and intersect the primary column.
    c. If neither the laying down nor overlaying in the previous two steps resulted in a non-NULL value being placed in any portion of the primary column, a NULL is inserted. Null indicators for intersecting redefines remain in place.

Additional Considerations

Consider the following carefully:

- While you are not prevented by CA Datacom/DB from defining DATACOM VIEWs that contain overlapping columns, we recommend against the use of overlapping columns when alternate solutions are possible. We are not responsible for the integrity of data or the operation of a DBMS that has been maintained through a view containing overlapping columns.
- When an array column is referenced as a single large CHAR column outside of any DATACOM VIEW, any default value for INSERT is applied once to the entire column. When referenced through a DATACOM VIEW, it is applied to each array element. This will change when view support is implemented outside of DATACOM VIEWs.