

# CA Datacom<sup>®</sup>/DB

## SQL User Guide

Version 14.02



This Documentation, which includes embedded help systems and electronically distributed materials (hereinafter referred to as the "Documentation"), is for your informational purposes only and is subject to change or withdrawal by CA at any time.

This Documentation may not be copied, transferred, reproduced, disclosed, modified or duplicated, in whole or in part, without the prior written consent of CA. This Documentation is confidential and proprietary information of CA and may not be disclosed by you or used for any purpose other than as may be permitted in (i) a separate agreement between you and CA governing your use of the CA software to which the Documentation relates; or (ii) a separate confidentiality agreement between you and CA.

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

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

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

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

The manufacturer of this Documentation is CA.

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

Copyright © 2015 CA. All rights reserved. All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

## CA Technologies Product References

This document references the following CA products:

- CA 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](mailto: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	21
System Tasks .....	21
Syntax Diagrams .....	21
CA Datacom/DB Extensions .....	21
Where to Find Information .....	21
Related Publications .....	21
Listing Libraries for CA Datacom Products .....	22
Sample Report Headers .....	22
Reading Syntax Diagrams .....	24
Statement Without Parameters .....	25
Statement with Required Parameters .....	25
Delimiters Around Parameters .....	25
Choice of Required Parameters .....	26
Default Value for a Required Parameter .....	26
Optional Parameter .....	26
Choice of Optional Parameters .....	27
Repeatable Variable Parameter .....	27
Separator with Repeatable Variable and Delimiter .....	27
Optional Repeatable Parameters .....	28
Default Value for a Parameter .....	28
Variables Representing Several Parameters .....	29
CA Datacom/DB Extensions .....	29
Chapter 2: Before You Start	31
What Is SQL? .....	31
What You Should Know About SQL .....	31
Tables .....	31
Columns .....	31
Rows .....	32
Views .....	32
Table and View Examples .....	32
Indexes .....	34
Cursors .....	34
Units of Work .....	34
Units of Recovery (Logical Unit of Work) .....	34
Isolation Levels .....	35

---

Schemas .....	37
Authorization ID .....	37
Accessor ID .....	37
Privileges .....	38
Synonym.....	38
SQL Statements .....	39
Binding .....	41
Plan .....	41
SQL Manager .....	42
Reserved Words .....	43

## Chapter 3: Getting Started 47

SQL Schemas .....	47
SQL Tables .....	47
SQL Tables and Logging.....	48
Creating SQL Tables.....	48
Using Existing Tables .....	49
Populating SQL Tables .....	50
Accessing SQL Tables.....	51
Selecting and Manipulating Data .....	51
Specifying Preprocessor Options.....	52
Preparing Programs.....	52
Mixed Mode Programming .....	53
Statement Execution Table .....	53
Dynamic SQL.....	55
Static SQL .....	56
Dynamic SQL .....	56
Dynamic SQL in CA Datacom/DB.....	57
INCLUDE Directive .....	57
Name Types.....	58
Reserved Words .....	58
Parameter Markers .....	58
Security Implications of Dynamic SQL.....	58
Using Dynamic SQL in Application Programs .....	58
Other Tasks.....	69
SQL Status Tables .....	70
Procedures and Triggers.....	70
Overview .....	71
SQL Procedures .....	74
External Security Support for Procedure/Trigger Creation and Execution .....	74
Trigger Execution for Record-at-a-Time Maintenance.....	75

---

Transaction Integrity .....	75
Subroutine Calls Inside Procedures.....	76
Restrictions .....	76
Multi-User Facility Considerations for Procedures .....	77
Parameter Styles and Error Handling.....	80
SQL Error Messages Related to Procedures and Triggers .....	83
Datadictionary Support for Triggers and Procedures .....	86
Examples: Creating a Procedure .....	87
Example: Calling a Procedure.....	109
Left Outer Joins .....	110
Overview of Joins .....	110
SELECT Statement Subselect Syntax .....	111
SELECT Statement Select-Into Syntax .....	112
Inner Join Example .....	112
Outer Join Example .....	112
Value of Rows That Do Not Match.....	113
WHERE Clause .....	114
Performance Considerations.....	115
Order of Predicate Evaluation.....	116
SQL Memory Guard.....	117
Activating the SQL Memory Guard .....	118
SQL and Multiple Multi-User Facilities Support .....	119
Application Design Considerations.....	119
Index-Only Processing .....	120
Cursor Processing.....	120
DBSQLPR.....	121
Processing .....	122
Line Commands.....	122
DBSQLPR Syntax .....	123
DBSQLPR Options.....	123
DROP PLAN (DBSQLPR) .....	132
Example JCL.....	133
Example SQL Statements .....	135
Sample Report.....	137
DATACOM VIEWS .....	138
Overview .....	138
Redefinitions .....	138
Arrays .....	141
Default Values for Redefinitions and Arrays .....	142
Datadictionary Considerations.....	143
Using SQC Table to Cancel SQL Requests.....	143
Overriding SQL Key Selection .....	143

---

---

Examples .....	145
XML Support.....	145
SQL Read-Only.....	146

## Chapter 4: CA Datacom/DB SQL Preprocessors 147

Input to the Preprocessor .....	148
INCLUDEs in COBOL.....	149
INCLUDEs in PL/I.....	149
INCLUDEs in Assembler .....	149
INCLUDEs in C.....	150
Output from the Preprocessors .....	150
COBOL .....	150
PL/I, Assembler, and C .....	151
Sample JCL.....	151
Sample COBOL JCL.....	152
Sample PL/I JCL .....	161
Sample Assembler JCL.....	163
Sample C Language JCL .....	168
Embedding SQL Statements in Host Programs .....	173
Distinguishing SQL Statements .....	173
Rules for Coding Embedded SQL.....	176
Coding Embedded SQL in COBOL.....	178
Coding Embedded SQL in PL/I.....	185
Coding Embedded SQL in Assembler .....	200
Coding Embedded SQL in C.....	206
Using Preprocessor Options.....	208
Overview .....	209
Naming the Plan.....	209
Specifying Processing Options in COBOL .....	210
Specifying Processing Options in PL/I, C, and Assembler.....	211
Options You Can Specify .....	214
Description of Options .....	217
SQL Communication Area (SQLCA).....	242
SQLCA in COBOL.....	242
SQLCA in PL/I.....	242
SQLCA in Assembler .....	243
SQLCA in C Language.....	243
Example SQLCA Formats .....	243
SQL Work Area (SQLWA) .....	261
SQLWA Examples .....	261
Error Handling .....	268



---

Interaction of Multiple Preprocessors .....	269
SQL Plan Options Special Topics.....	270
Read-Only.....	270
Mixing Isolation Levels .....	270
Locking a Row.....	270
CICS Unit of Recovery End.....	271
ANSI Compatibility .....	272
CA Ideal Considerations .....	273
Block Transfer .....	273
OPEN/CLOSE Efficiency .....	273
Automatic Unit of Recovery End.....	273
Plan Locks.....	274
<b>Chapter 5: Interfacing with the User Requirements Table (URT)</b> .....	<b>277</b>
DBURINF - User Requirements Interface .....	278
DBURSTR - Start User Requirements Table .....	279
DBUREND - End Interface/Table .....	279
Example.....	279
<b>Chapter 6: Program Compilation, Link-Edit and Execution</b> .....	<b>281</b>
Batch Link-Editing and Execution .....	281
Linking Multiple Modules with SQL.....	281
Sample JCL for Batch .....	284
CICS Link-Editing and Execution .....	284
Sample JCL for CICS .....	285
IMS/DC Link-Editing and Execution .....	285
Sample JCL for IMS/DC.....	285
z/OS IMS/DC Sample JCL.....	286
<b>Chapter 7: SQL Error Handling</b> .....	<b>289</b>
SQL Return Codes -117 and -118.....	289
Online Displays.....	292
Batch Output.....	296
Error Handling Related to Procedures and Triggers.....	298
SQL States.....	298
SQLCA Examples.....	298
SQL State Classes.....	306
SQL States Table.....	308

---

Chapter 8: Application Tasks Using Embedded SQL	317
Chapter 9: Specifying Result Tables	321
Selecting All Columns	321
Selecting Some Columns	322
Selecting Using Search Conditions	323
Ordering by Column Values	324
Eliminating Duplicate Rows	325
Counting	326
Calculating Values	327
Summarizing Group Values	329
Testing for Existence	331
Chapter 10: Selecting Data from Multiple Tables	333
Joining Tables	334
Using the UNION Operator	337
Chapter 11: Inserting Rows	341
Chapter 12: Updating a Table	343
Chapter 13: Deleting Rows	347
Chapter 14: Committing and Backing Out Transactions	349
Chapter 15: Overview of the Interactive SQL Service Facility	351
Chapter 16: Using the Interactive SQL Service Facility	355
Executable SQL Statements	356
Specifying Unique SQL Names	356
Submitting SQL Statements	357
How to Submit SQL Statements	358
How to Use	359
Using Commands	366
Commands Specifically for Use in the Interactive SQL Service Facility	368
ALTERNATE Command	368
COPY SQL Command	368
DELETE SQL Command	370

---

DISPLAY SQL Command.....	371
EDIT SQL Command.....	372
EXECUTE Command .....	373
REBIND Command.....	373
SCROLL Command .....	374
Using Line Commands .....	375
Using Margin Commands .....	376
Using PF Keys.....	377
Maintaining Source and Output Members .....	379
Editing and Executing Source Members .....	380
Displaying Source and Output Members .....	388
Copying Source Members .....	393
Deleting Source and Output Members .....	395

## Chapter 17: Creating SQL Objects 399

Creating a Schema.....	400
Naming the Schema .....	400
Relating the Person to the AUTHID.....	401
Changing Your AUTHID.....	402
System Schemas.....	402
Displaying and Reporting .....	403
Example Source Member .....	403
Example Output Member .....	405
Creating a Table .....	405
Naming the Table.....	406
Key Creation.....	407
Element Creation .....	408
Statement Execution Results .....	409
Example Source Member .....	409
Example Output Member .....	411
Altering a Table .....	412
Statement Execution Results .....	413
Example Source Member .....	413
Example Output Member .....	414
Creating an Index .....	415
Naming the Index (Key).....	416
Key Creation.....	416
Statement Execution Results .....	416
Example Source Member .....	417
Example Output Member .....	418
Creating a View .....	419

---

Naming the View .....	419
Example Source Member .....	420
Example Output Member .....	422
Creating a Synonym .....	423
Naming the Synonym .....	424
Example Source Member .....	425
Example Output Member .....	426
Adding and Replacing Comments .....	427
Example Source Member .....	427
Example Output Member .....	428
Chapter 18: Deleting SQL Objects .....	431
Deleting a Schema .....	432
Dropping an Index .....	432
Dropping a Table .....	434
Dropping a View .....	437
Dropping a Synonym .....	439
Chapter 19: Manipulating Data in SQL Tables .....	443
Chapter 20: Controlling Access Through SQL Statements .....	445
Chapter 21: Performing SQL Administrative Tasks .....	447
SQL Names .....	447
Setting the Session Authorization ID .....	448
Current Authorization ID at Start of Session .....	448
Displaying and Reporting .....	449
How to Set the Default .....	450
Deleting a Plan .....	452
How the Plan Is Named .....	453
How to Delete a Plan .....	454
Rebinding a Plan .....	457
How to Rebind a Plan .....	457
Displaying Index of SQL Plans .....	460
How to Display an Index of SQL Plans .....	461
Specifying Plan Options in a Source Member .....	463
Coding Plan Options .....	464
Options You Can Specify .....	467
Example .....	473

---

Chapter 22: Overview of SQL Language Reference	475
--	-----

Chapter 23: Basic Language Elements	477
-------------------------------------	-----

Characters .....	477
Tokens .....	477
Spaces .....	478
Uppercase and Lowercase .....	478
Identifiers .....	478
Ordinary SQL identifiers .....	479
Delimited SQL identifiers .....	479
Naming Conventions .....	480
Authorization ID .....	483
Values .....	484
Data Types .....	485
DATE, TIME, and TIMESTAMP .....	485
Host Variable Data Types .....	486
SQL Data Types .....	486
CA Datacom/DB Data Types .....	486
SQL Data Type Support for All CA Datacom/DB Tables .....	490
Character Strings .....	495
Numeric Data Types .....	500
Basic Operations (Assignment and Comparison) .....	501
Numeric Assignments .....	502
String Assignment .....	505
Numeric Comparisons .....	507
String Comparisons .....	509
Literals .....	510
Character String Literals .....	510
Integer Literals .....	512
Floating Point Literals .....	513
Decimal Literals .....	513
Column Names .....	514
Qualified Column Names .....	515
Correlation Names .....	515
Column-Name Qualifiers to Avoid Ambiguity .....	517
Column-Name Qualifiers in Correlated References .....	518
Host Variables .....	520
Host Structures .....	520
Extended Format for Host Variables in COBOL .....	522
Indicator Variables .....	524
SQL Parameters .....	525

---

SQL Variables.....	525
<b>Chapter 24: Expressions</b>	<b>527</b>
CASE, COALESCE, NULLIF, and CAST .....	529
Special Registers.....	533
Labeled Duration.....	535
Expressions without Arithmetic Operators .....	537
Expressions with the Concatenation Operator .....	537
Expressions with Arithmetic Operators .....	538
Arithmetic Operations for Dates, Times, and Timestamps .....	542
Durations.....	542
Precedence of Operations.....	546
<b>Chapter 25: Functions</b>	<b>549</b>
Column Functions.....	549
Description .....	549
Rules for Column Functions .....	551
Examples .....	553
Scalar Functions .....	554
Rules for Scalar Functions .....	555
Description .....	555
Character Functions .....	565
Bit-Level Functions .....	570
Byte-Level Function.....	573
XML Functions.....	574
<b>Chapter 26: Predicates</b>	<b>581</b>
Basic Predicate .....	581
Quantified Predicate .....	583
BETWEEN Predicate .....	584
LIKE Predicate.....	585
EXISTS Predicate .....	589
IN Predicate .....	590
NULL Predicate .....	592
<b>Chapter 27: Search Conditions</b>	<b>593</b>
<b>Chapter 28: SQL Statements</b>	<b>597</b>
Preliminary Information—Lock Levels .....	597

---

Statements That Support Procedures and Triggers .....	597
ALTER TABLE .....	598
Description .....	601
Processing .....	606
Assignment Statement .....	610
CALL/EXECUTE PROCEDURE .....	610
CASE Statement .....	612
CLOSE .....	612
Description .....	613
Processing .....	613
COMMENT ON .....	613
Description .....	614
COMMIT WORK .....	616
Description .....	616
CREATE INDEX .....	618
Description .....	619
Processing .....	620
CREATE PROCEDURE .....	620
External Procedures .....	622
SQL Procedures .....	622
CREATE PROCEDURE Syntax and Description .....	624
CREATE RULE .....	676
CREATE SCHEMA .....	676
Description .....	677
CREATE SYNONYM .....	678
Description .....	679
CREATE TABLE .....	680
Description .....	681
Privileges .....	682
Column Definition .....	683
Column Constraint Definition .....	684
Table Constraint Definition .....	687
Referential Constraint Definition .....	689
Data Types .....	695
CREATE TRIGGER/RULE .....	702
CREATE VIEW .....	705
Privileges .....	706
Description .....	707
Processing .....	708
DECLARE CURSOR .....	710
Description .....	712
Cursor Usage .....	714

---

Example1.....	715
Example 2.....	716
Example 3.....	716
DECLARE STATEMENT .....	717
DELETE.....	717
Searched DELETE.....	718
Positioned DELETE.....	718
Description .....	719
Processing .....	720
DESCRIBE .....	721
Description .....	723
DROP .....	725
Description .....	728
Example 1.....	731
Example 2.....	731
Example 3.....	731
EXECUTE .....	731
Description .....	732
Parameter Marker Replacement .....	733
EXECUTE IMMEDIATE.....	734
Description .....	735
Rules for Statement Strings.....	735
EXECUTE PROCEDURE .....	735
FETCH .....	736
Description .....	738
GRANT .....	742
Plan Security.....	742
Description of Plan Security Diagram.....	743
Description of Non-Plan Security Diagram.....	744
IF-THEN Statement.....	746
INSERT .....	747
Description .....	748
Rules for Inserting .....	749
Processing .....	749
ITERATE Statement.....	750
LEAVE Statement.....	750
LOCK TABLE .....	751
Description .....	751
Example.....	752
LOOP Statement.....	752
OPEN .....	752
Description .....	753



---

Processing .....	755
Effect of Temporary Tables .....	755
Example .....	755
PREPARE .....	756
Description .....	757
Rules for Statement Strings.....	758
Rules for Parameter Markers .....	758
REPEAT-UNTIL Statement.....	759
REVOKE.....	759
Plan Security.....	760
Description of Plan Security Diagram.....	760
Description of Non-Plan Security Diagram.....	762
Example 1.....	763
Example 2.....	764
Example 3.....	764
ROLLBACK WORK.....	764
Description .....	765
SELECT .....	766
Subselect .....	767
Full-Select Statement .....	776
Description .....	776
Select-Statement.....	777
ORDER BY Clause.....	779
Select-Into Statement .....	781
SET CURRENT SQLID .....	787
Description .....	788
Example.....	789
UPDATE .....	789
Searched UPDATE .....	790
Positioned UPDATE .....	790
WHENEVER.....	794
Description .....	794
Processing .....	795
Example.....	796
WHILE Statement .....	796

## Appendix A: SQL Query Optimization Messages 797

Message Table (SYSADM.SYSMSG) .....	797
Requesting Messages .....	798
Bind-time Messages .....	798
Bind-time Summary Messages.....	798

---

Bind-time Detail Messages.....	807
Execution-Time Messages.....	812
Execution-Time Summary Messages.....	813
Execution-Time Detail Messages.....	816
Examples.....	818
Appendix B: Accessibility Features	821
Appendix C: Sample Data Tables	823
CUSTOMERS Table: Sample Data.....	824
ORDERS Table: Sample Data.....	826
Appendix D: Results of Defining Structures Using SQL Statements	827
CREATE INDEX Statement.....	827
CREATE PROCEDURE Statement.....	828
CREATE SCHEMA Statement.....	829
CREATE TABLE Statement.....	829
CREATE SYNONYM Statement.....	837
CREATE TRIGGER Statement.....	837
CREATE VIEW Statement.....	839
Appendix E: Results of Using ALTER TABLE	841
Appendix F: SQL Object Consistency Analyzer and Upgrade Rebind Utilities	849
SQL Object Consistency Analyzer.....	849
Running the SQL Object Consistency Analyzer.....	850
Correcting Problems.....	856
DBSRFPR (SQL Upgrade Rebind Utility).....	861
Rebinding with DBSRFPR.....	861
Dropping Plans with DBSRFPR.....	865
Sample JCL.....	867
Appendix G: SQL Descriptor Area (SQLDA)	869
Determining Number of SQLVAR Entries to Use.....	869
SQLDA (DESCRIBE or PREPARE INTO Statements).....	870
Data Type of the Column and Null Value.....	871
Data Type of the Column.....	872
SQLVAR and SQLTYPE.....	873
SQLDA (EXECUTE, FETCH, or OPEN Statement).....	873

---

SQLVAR and VS/COBOL.....	875
--------------------------	-----



# 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 24).

## CA Datacom/DB Extensions

For information on CA Datacom/DB extensions to ANSI standard SQL implementations, see [CA Datacom/DB Extensions](#) (see page 29).

## 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 guide. 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 z/OS, the default name for this library is CABDMAC). In z/VSE, sample PROCs are provided that allow you to use parameter substitution. You can copy and modify these samples for your specific requirements.

Code JOB statements to your site standards and specifications. Specify all data set names and library names 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
3. CA Common Services for z/OS base libraries (*hlq.CAW0LOAD*)
4. CA IPC libraries (*hlq.CAVQLOAD*)
5. Libraries for additional products, such as [assign the DCS variable value for your book], CA Datacom VSAM Transparency, CA Ideal, and so on

## Sample Report Headers

The report headers for the sample reports contained in this guide are shown here.

DBSQLPR SQL Processor report headers have the following format:

```
Date: mm/dd/ccyy ***** Page: 1
* CA Datacom/DB SQL Option *
Time: hh.mm.ss * DBSQLPR SQL Processor * Version: 14.0
* Copyright © 1990-2011 CA. All rights reserved. *
*****
```

SQL Object Consistency Analyzer report headers have the following format:

```

*****
*      CA-DATACOM/DB SQL OBJECT CONSISTENCY ANALYZER      *
*      Copyright C 1990-2011 CA. All rights reserved.      *
*      VERSION 14.0                                       *
*****
Page: 1
Date: mm/dd/ccyy
Time: hh.mm.ss

```

General Utility report headers have the following format:

```

Date: mm/dd/ccyy ***** Page: 1
*      CA Datacom/DB *
Time: hh.mm.ss *      General Utility *      Version: 14.0
*      Copyright © 1990-2011 CA. All rights reserved. *
Base: dbid ***** Directory: name

```

**Base:**

The DATACOM-ID (DBID) of the database (base) in use when the report was assembled is shown.

**Note:** Base does not appear in the header If not appropriate to the report that was generated.

**Date:**

The date when the report was assembled is shown in the format mm/dd/ccyy:

**mm**

month

**dd**

day

**cc**

century

**yy**

year

**Directory:**

The name of the Directory (CXX) in use when the report was assembled is shown.

**Note:** Directory does not appear in the header If not appropriate to the report that was generated.

**Page:**

The number of the page of the report.

**Time:**

The time when the report was assembled is shown in the format hh.mm.ss:

**hh**

hour

**mm**

minutes

**ss**

seconds

**Version:**

The version of CA Datacom/DB being executed when the report was assembled is shown, for example, Version 14.0.

## 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

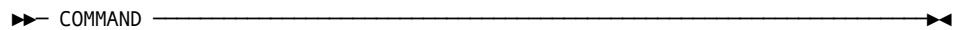
---



)	close parenthesis	¬	not sign
+	addition	-	subtraction
*	multiplication	/	division

## Statement Without Parameters

The following is a diagram of a statement without parameters:



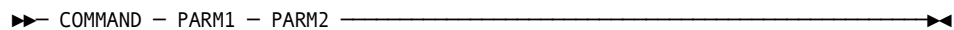
For this statement, you must write the following:

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:



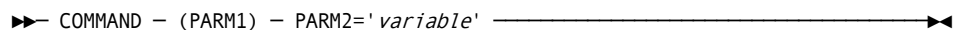
You must write the following:

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:



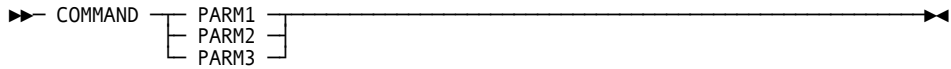
If the word *variable* is a valid entry, you must write the following:

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:



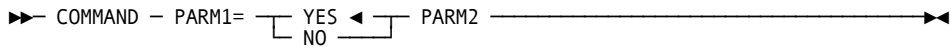
You can choose one of the parameters from the vertical list, such as in the following examples:

```
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:



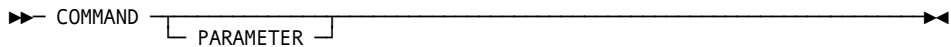
If you specify the command, you must write one of the following:

```
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:



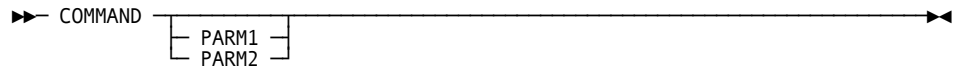
You can choose (or not) to use the optional parameter, as shown in the following examples:

```
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:

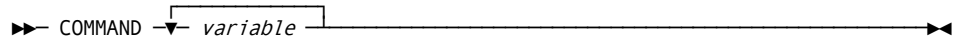
```

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:

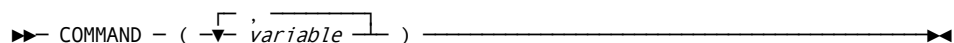
```

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:



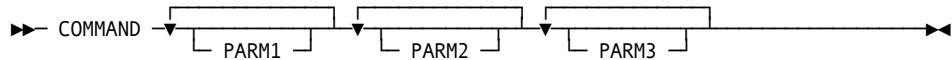
In the preceding diagram, the word *variable* is in lowercase italics, indicating that it is a value you supply. It is also on the main path, which means that you must specify at least one entry. The repeat symbol indicates that you can specify more than one variable and that you must separate the entries 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:



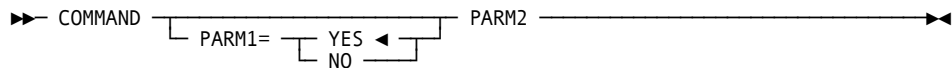
The following are some of the statements you can write:

```
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:

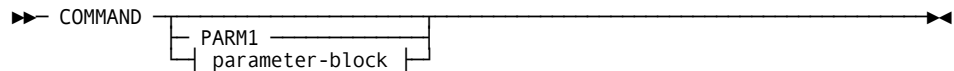


For this command, COMMAND PARM2 is the equivalent of COMMAND PARM1=YES PARM2.

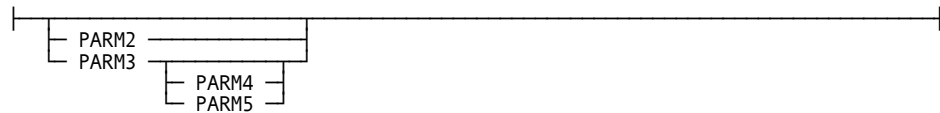
## Variables Representing Several Parameters

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
  
```

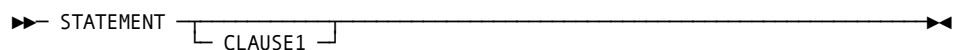
**Note:** Before you can specify PARM4 or PARM5 in this command, you must specify PARM3.

## 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:

	<b>EMPNO</b>	<b>LNAME</b>	<b>FNAME</b>	<b>MI</b>	<b>CITY</b>	<b>ST</b>
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:

	<b>EMPNO</b>	<b>SALARY</b>	<b>YTDCOM</b>
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:

	<b>LNAME</b>	<b>FNAME</b>	<b>MI</b>	<b>SALARY</b>
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 (Logical Unit of Work)

A unit of recovery also known as a Logical Unit of Work (LUW) 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 53).

CA Datacom/DB supports the dynamic preparation and execution of SQL statements under the control of an application program. See [Dynamic SQL](#) (see page 55).

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:

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

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

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

See the descriptions of the SQL statements beginning with [ALTER TABLE](#) (see page 598) 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.

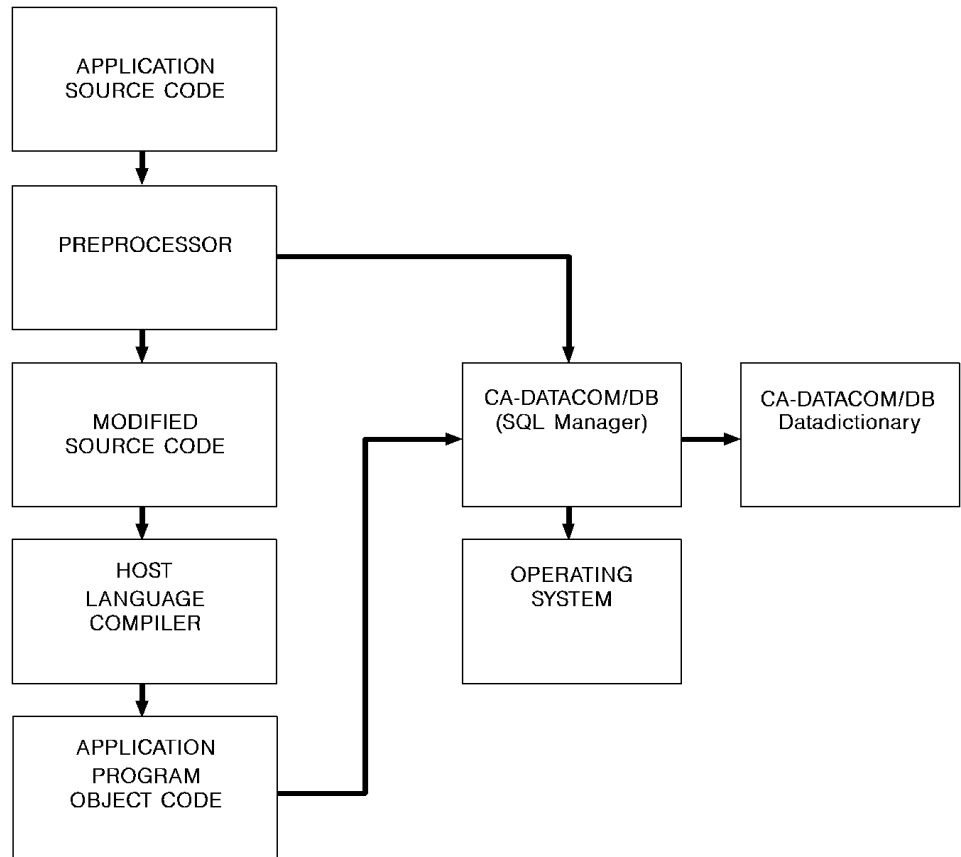
Since SQL plans are stored in the CA Datacom Datadictionary, the CA Datacom Datadictionary must be available to execute previously prepared SQL statements.

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 Datadictionary 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			
INEXTRACT			
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 138).
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 138)), 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](#) (see page 173))
- Submit through CA Datacom Datadictionary Interactive SQL Service Facility (see [Using the Interactive SQL Service Facility](#) (see page 355))
- 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](#) (see page 146).

## 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 USRNTY=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	*	*	*

<b>SQL Statement</b>	<b>CA Datacom Datadictionary Interactive SQL Service Facility (Interactive)</b>	<b>In an application program prepared using an CA Datacom/DB SQL Preprocessor (Embedded)</b>	<b>CA Dataquery (SQL &amp; Batch Modes)</b>
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 inflexible. 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](#) (see page 717))  
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](#) (see page 721))  
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](#) (see page 731))  
This statement executes an SQL statement that has previously been prepared for execution by the PREPARE statement.
- **EXECUTE IMMEDIATE** (see [EXECUTE IMMEDIATE](#) (see page 734))  
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](#) (see page 756))  
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](#) (see page 752))

## 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 \(SQLDA\)](#) (see page 869)). 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](#) (see page 194), or for Assembler see [Rules for SQL INCLUDEs in Assembler](#) (see page 203). For an example SQLDA in COBOL, see [Example](#) (see page 724).

## Name Types

Two CA Datacom/DB name types (descriptor-name and statement-name) support dynamic SQL. See [Naming Conventions](#) (see page 480) for more information.

## Reserved Words

Four reserved words pertain to dynamic SQL, as shown on [Reserved Words](#) (see page 43): 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](#) (see page 756).

## 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 61)).
- 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 61)).
- 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 62)).
- 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 64).

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

## Using DBXHAPR in Dynamic SQL

DBXHAPR is only for COBOL programs using dynamic SQL. COBOL does not allow the SQLDATA and SQLIND fields to be declared as pointer variables. When you nevertheless need to set them to the address of host variables in SQLVAR, follow these steps:

1. Link DBXHAPR to the OBJLIB that is being used by adding the following to your COBOL programs:  
`CALL 'DBXHAPR' USING <SQLDATA>, <host-var>.`
2. Set the address of each SQLDATA and SQLIND field used before SQLDA with the following in the linkedit step:  
`INCLUDE OBJLIB(DBXHAPR)`

---

## 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 61)).
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 `SMSQLDA` 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 ( $16 + \text{SQLD} * 44$ ). 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](#) (see page 620).

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](#) (see page 620))
- CALL/EXECUTE PROCEDURE —without SET processing support (see [CALL/EXECUTE PROCEDURE](#) (see page 610))
- DROP PROCEDURE (see [DROP](#) (see page 725))
- CREATE TRIGGER/RULE—row level only (see [CREATE TRIGGER/RULE](#) (see page 702))
- DROP TRIGGER/RULE (see [DROP](#) (see page 725))

For examples showing the use of procedures and triggers, see [Examples: Creating a Procedure](#) (see page 87) and [Example: Calling a Procedure](#) (see page 109).

**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](#) (see page 620).

## 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](#) (see page 610) 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](#) (see page 298).

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 Reerii, 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, reprocess, and recompile the procedure to follow all instructions given in [Parameter Styles and Error Handling](#) (see page 80).

-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 108))

### 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 for z/OS and [Sample JCL for z/VSE](#) (see page 106) 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 75), [Restrictions](#) (see page 76), and [Parameter Styles and Error Handling](#) (see page 80).

**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.
//TWOUP  OUTPUT  DEFAULT=YES,FORMDEF=010111,PAGEDEF=W120C0,CHARS=(GT20)
+INC GRB.JOBLIB3                                0000010
+INC GRB.EDCC
+INC GRB.CEEVARS
//* *****
//* * "C" PRECOMPILE STEP ***
//* *****
//CPRECOMP EXEC PGM=DBPLIPR,PARM='PLANNAME=ITEMKILL'
//PROCLIB DD DSN=CA90SMVS.NEW.C.R3V1.PROCLIB,DISP=SHR
//SOURCE DD DSN=DCMDEV.SQL.GARBR02.SRCLIB2(ITEMKILL),DISP=SHR
//SYSUDUMP DD SYSOUT=*
//REPORT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSOUT DD SYSOUT=*
//OPTIONS DD DSN=DCMDEV.SQL.GARBR02.SRCLIB2(TPRCCOPT),DISP=SHR
//SRCOUT DD DSN=&&SRC,DISP=(,PASS,DELETE),UNIT=VIO,
//          SPACE=(2000,(200,200)),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//* *****
//*
//COMPA EXEC PROC=EDCC,          (FROM SYS2.PROCLIB)
//  CRUN='RENT',
//  CPARM='NOMARGINS,NOSEQUENCE,LIST,SOURCE',
//  CPARM2='LOCALE("POSIX"),LANGVLV(ANSI),OMVS,DLL',
//  CPARM3='SSCOM,LONGNAME,SHOWINC',
//  INFILE='&&SRC',
//  OUTFILE='DCMDEV.SQL.GARBR02.OBJLIB(ITEMKILO)'
//*
//SYSLIB DD DSN=CEE.SCEEH.H,DISP=SHR
//          DD DSN=CEE.SCEEH.SYS.H,DISP=SHR
//USERLIB DD DSN=DCMDEV.SQL.LIBRMAS,DISP=SHR,SUBSYS=LAM
//*
//* SYSCPRT DD DSN=DCMDEV.SQL.PRINT(TCPG002),DISP=SHR
//*
//STEP1 EXEC PGM=DBUTLTY,REGION=2048K,COND=EVEN
//PXX DD DSN=DCMDEV.DB.MUF3.PXX,DISP=SHR
//CXX DD DSN=DCMDEV.DB.MUF3.CXX,DISP=SHR
//SNAPER DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSIN DD *
REPORT AREA=PXX,DUMPS=FULL
/*
//* ***** LINK STARTS HERE *****
//PRELINK EXEC PGM=EDCPRLK,
//  PARM='POSIX(OFF)/OE,MEMORY,DUP,NOER,MAP,NOUPCASE,NONCAL'
//*
//SYMSGS DD DSN=CEE.SCEEMSGP(EDCPMSGE),DISP=SHR

```

```
//OBJLIB DD DSN=DCMDEV . SQL . GARBR02 . OBJLIB , DISP=SHR
//C8941 DD DSN=CEE . SCCEOBJ , DISP=SHR
//SYSOUT DD SYSOUT=*
//SYSPRINT DD *
//SYSMOD DD DSN=DCMDEV . SQL . GARBR02 . OBJLIB ( ITEMKOBJ ) , DISP=SHR
//SYSDEFSD DD DUMMY
//SYSIN DD *
    INCLUDE OBJLIB ( ITEMKILO )
/*
//LINKEDIT EXEC PGM=LINKEDIT ,
// PARM=( ' AMODE=31 , RMODE=ANY , TERM=YES , MSGLEVEL=0 , MAP , DYNAM=DLL ' ,
// ' CALL=YES , CASE=MIXED , REUS=RENT , EDIT=YES ' )
//SYSLIB DD DSN=CEE . SCEELKED , DISP=SHR
// DD DSN=SYS1 . CSSLIB , DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTEM DD SYSOUT=*
//SYSLMOD DD DISP=SHR , DSN=DCMDEV . SQL . GARBR02 . LODLIB2 ( ITEMKILL )
//OBJLIB DD DSN=DCMDEV . SQL . GARBR02 . OBJLIB , DISP=SHR
// *CALIB DD DSN=DCMDEV . DB . R100 . LODLIB , DISP=SHR
//CALIB DD DSN=DCMDEV . DB . R100 . LODLIB , DISP=SHR
//CEELIB DD DSN=DCMDEV . DBDT . DSYTEST . LOADLIB , DISP=SHR
//SYSLIN DD *
    INCLUDE OBJLIB ( ITEMKOBJ )
    INCLUDE CEELIB ( CEEUOPT )
    INCLUDE CALIB ( DBXPIPR )
/*
// *+INC GRB . URTCEE
```

```

                /***** 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 128;
    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_PARAMS_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 // CPARAM='NOMARGINS,NOSEQUENCE,LIST,SOURCE',
000213 // CPARAM2='LOCALE("POSIX"),LANGVL(ANSI),OMVS,DLL',
000214 // CPARAM3='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.LIBRMAS,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 //SYMSGS DD DSN=CEE.SCEEMSGP(EDCPMSG),DISP=SHR
000226 //OBJLIB DD DSN=dsname.OBJLIB,DISP=SHR
000227 //C8941 DD DSN=CEE.SCEE0BJ,DISP=SHR
000228 //SYSOUT DD SYSOUT=*
000229 //SYSPRINT DD *
000230 //SYSMOD DD DSN=dsname.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 //SYSTEM DD SYSOUT=*
000242 //SYSMOD DD DISP=SHR,DSN=dsname.LODLIB2(ITEMKILL)
000243 //OBJLIB DD DSN=dsname.OBJLIB,DISP=SHR
000244 //CAILIB DD DSN=xxxxxx.xx.xxxx.LODLIB,DISP=SHR
000245 //CEELIB DD DSN=xxxxxx.xxxx.xxxxxx.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). For a z/VSE sample in COBOL, see [Sample JCL for z/VSE](#) (see page 106).

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.
//              CLASS=A,MSGCLASS=X,REGION=2048K
//JOBLIB       DD DSN=library-containing-DBSIDPR,DISP=SHR
//              DD DSN=library-containing-multi-user-modules,DISP=SHR
//              DD DSN=etc...,DISP=SHR
//*-----*
//* STEP 1: PRE-COMPILE THE PROCEDURE PROGRAM                *
//*-----*
//PRECOMP     EXEC PGM=DBXMMPR
//WORK1       DD DSN=&. &WORK1.,UNIT=SYSDA,DISP=(NEW,PASS),
//              DCB=(RECFM=F,LRECL=80,BLKSIZE=80),SPACE=(TRK,(1,1))
//WORK2       DD DSN=&. &WORK2.,UNIT=SYSDA,DISP=(NEW,PASS),
//              DCB=(RECFM=F,LRECL=80,BLKSIZE=80),SPACE=(TRK,(1,1))
//WORK3       DD DSN=&. &WORK3.,UNIT=SYSDA,DISP=(NEW,PASS),
//              DCB=(RECFM=F,LRECL=80,BLKSIZE=80),SPACE=(TRK,(1,1))
//SYSOUT      DD SYSOUT=*
//SYSPRINT    DD SYSOUT=*
//SNAPER      DD SYSOUT=*
//SNAPER      DD SYSOUT=*
//SYSPUNCH    DD DSN=&. &SQLCOB.,UNIT=SYSDA,DISP=(NEW,PASS),
//              DCB=(RECFM=F,LRECL=80,BLKSIZE=80),SPACE=(TRK,(1,1))
//SYSUDUMP    DD SYSOUT=*
//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.
```

```

*$DBSQLOPT PROCSQLUSAGE=MODIFIES USRNTY=NONE
*$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-DBC-EXT TO DBCODE-EXT.
        MOVE SQLCA-DBC-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.          *****

```

```
/*
/**      *** End of Program and Continuation of JCL. ***
/**

/**-----*
/** STEP2: COMPILE COBOL USER PROGRAM OUTPUT FROM COBOL PRECOMPILER  *
/**-----*
/**
/**COBOL EXEC PGM=IGYCRCTL,
/**      PARM='RENT,NUM,NODYN,APOST,NOSEQUENCE,LIST',
/**      COND=(8,LT)
/**SYSLIN DD DISP=(MOD,PASS),DSN=&.&COBOLOD.,
/**      UNIT=SYSDA,SPACE=(TRK,(15,15))
/**SYSPRINT DD SYSOUT=*
/**SYSUT1 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
/**SYSUT2 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
/**SYSUT3 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
/**SYSUT4 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
/**SYSUT5 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
/**SYSUT6 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
/**SYSUT7 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
/**SYSIN DD DSN=&.&SQLCOB.,UNIT=SYSDA,DISP=(OLD,DELETE,DELETE)
/**
```



```
/*-----*
/* STEP3: LINK USER PROGRAM WITH SYSTEM MODULES
*

/*-----*
/*
//LINK EXEC LKED,COND=(8,LT),
// PARM.LKED='RENT,XREF,LIST,LET,MAP'
//LKED.SYSLIN DD DSN=&.&COBOLOD.,UNIT=SYSDA,DISP=(OLD,DELETE,DELETE)
// DD DDNAME=SYSIN
//LKED.SYSLMOD DD DSN=dsname.LODLIB2,DISP=SHR
//LKED.SYSLIB DD DSN=CEE.SCEELKED,DISP=SHR
//LKED.OBJLIB DD DSN=DCMALL.R900.CAILIB,DISP=SHR
// DD DSN=DCMDEV.DB.R100.TST.LODLIB,DISP=SHR
// DD DSN=DCMDEV.DB.R100.LODLIB,DISP=SHR
//LKED.CEELIB DD DSN=DCMDEV.DBDT.DSYTEST.LOADLIB,DISP=SHR
//LKED.SYSIN DD *
INCLUDE CEELIB(CEEUOPT)
INCLUDE OBJLIB(DBXHVPR)
INCLUDE OBJLIB(DBXPIPR)
NAME ITEMKILL(R)
/*
```

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. For sample C JCL, see Sample JCL for C.

**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.

```
* $$ JOB ...           See the note above.
* $$ LST ...
// JOB name
* *****
* *
* * PRECOMPILE COBOL SQL
* *
* *****
// EXEC PROC=procname
// DLBL IJSYSPH, 'sql.syspunch',0,SD
// EXTENT SYSPCH,volser,1,0,r,n
ASSGN    SYSPCH,DISK,VOL=volser,SHR
// ASSGN  SYSnnn,DISK,VOL=volser,SHR
// DLBL   WORK1, 'name.precomp.work1',000,SD
// EXTENT SYSnnn,volser,1,0,r,n
// DLBL   WORK2, 'name.precomp.work2',000,SD
// EXTENT SYSnnn,volser,1,0,r,n
// DLBL   WORK3, 'name.precomp.work3',000,SD
// EXTENT SYSnnn,1,0,r,n
// EXEC DBXMMPR,SIZE=1024K
CBL LIB,NOMAP,APOST

      *COBOL SOURCE CODE GOES HERE*

/*
```

```
CLOSE SYSPCH,PUNCH
// IF $RC GT 0 THEN
// GOTO EOJ
* ****
* *
* * COBOL COMPILER EXECUTION *
* *
* ****
// DLBL IJSYSIN,'name.precomp.outfile',0,SD
// EXTENT SYSIPT,volser
ASSGN SYSIPT,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 IJSYS01,'name.workfile.SYS001',0,SD
// EXTENT SYS001,volser,1,0,r,n
// DLBL IJSYS02,'name.workfile.SYS002',0,SD
// EXTENT SYS002,volser,1,0,r,n
// DLBL IJSYS03,'name.workfile.SYS003',0,SD
// EXTENT SYS003,volser,1,0,r,n
// DLBL IJSYS04,'name.workfile.SYS004',0,SD
// EXTENT SYS004,volser,1,0,r,n
// DLBL IJSYS05,'name.workfile.SYS005',0,SD
// EXTENT SYS005,volser,1,0,r,n
// DLBL IJSYS06,'name.workfile.SYS006',0,SD
// EXTENT SYS006,volser,1,0,r,n
// DLBL IJSYS07,'name.workfile.SYS007',0,SD
// EXTENT SYS007,volser,1,0,r,n
// OPTION CATAL
  PHASE phasename,*
// EXEC IGYCRCTL,SIZE=1024K
/*
CLOSE SYSIPT,SYSRDR
/*
```

```
// IF $RC GT 4 THEN
// GOTO E0J
* *****
* *
* * LINK EDIT STEP *
* *
* *****
INCLUDE DBXHVPR
INCLUDE DBXPIPR
ENTRY BEGIN
// EXEC LNKEDT,SIZE=1024K
/. E0J
/*
// EXEC LISTLOG
/&
* $$ E0J
```

## 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

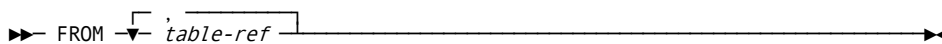
### Overview of Joins

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.

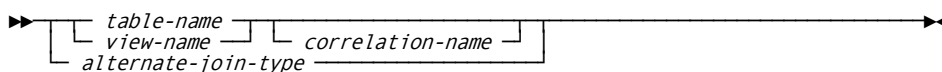
There are different kinds of joins. *Inner joins* eliminate from the resulting table the combined rows that do not satisfy evaluation against the join predicates. Therefore, with inner joins if no matching row is found, no rows are returned. Inner joins were supported by CA Datacom/DB in previous versions 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).

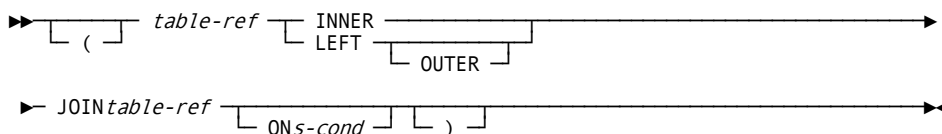
In the following diagrams, while a table reference (table-ref) is shown to be the main component of the FROM clause, it can also be referenced directly from inside the JOIN syntax (third diagram, alternate-join-type syntax).



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:

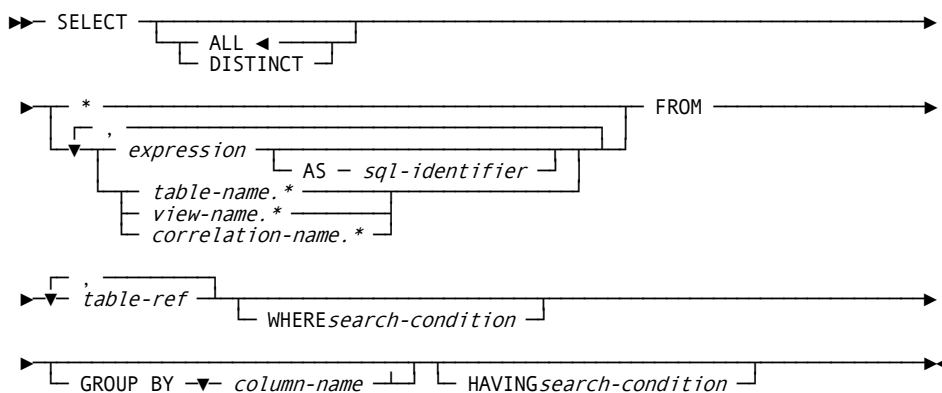


**Note:** The *s-cond* (search-condition) specified in the optional ON clause differs from the one in the WHERE clause in that the ON clause defines the join conditions that determine which rows contain nulls, as opposed to the WHERE clause, which eliminates rows from the result entirely. Also note that 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.

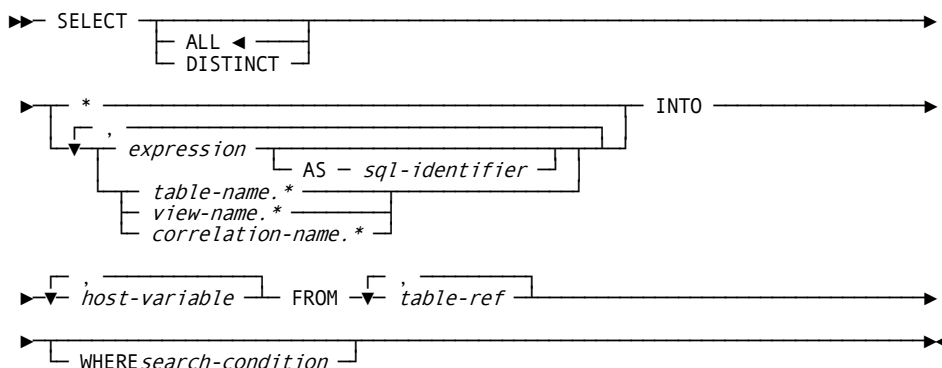
## SELECT Statement Subselect Syntax

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 116)

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 condition is 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.
- ISOLEVEL=R is not used, because isolation level R prevents index-only processing.

**Note:** VARCHAR columns with lengths that include trailing blanks have length set to exclude the trailing blanks, because VARCHAR lengths are not stored in the index.

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 SQL 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 123)).
- 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 123).
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](#) (see page 210).

3. The SYSIN file is read, and the input is processed as a series of commands (see [Line Commands](#) (see page 122)) 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](#) (see page 725) and DROP FUNCTION statements to drop procedures.

For more information about DROP PLAN, see [DROP PLAN \(DBSQLPR\)](#) (see page 132).

--

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)**

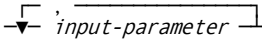
Prints output in tabular form, reverting to \*\$THIN if data exceeds PRTWIDTH= specification.

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

Zeroes the job-step return code.

## DBSQLPR Syntax

Following is the syntax for DBSQLPR:

```
▶▶ EXEC PGM=DBSQLPR - ,PARM='  ' ▶▶
```

**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](#) (see page 210).

**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

**PAGELN=**

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:**

DBSQLx, 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:**

DBSQLx, 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.

**TBLHRRPT=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](#) (see page 725) 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 122)).

►► DROP PLAN *auth-id.* *plan-name* ◀◀

#### ***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, and parameter separators in the PARM= input parameter string must be spaces.

Following is an example PARM= input parameter string specification.

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

The PARM= input parameter string is limited by IBM to 100 bytes. Following is an example in which lines have been spanned. Be aware that 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'
```

Following is a z/OS JCL sample:

**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.  
//             CLASS=K,MSGCLASS=X,REGION=1024K  
//SQLEXEC EXEC PGM=DBSQLPR,...see Listing Libraries for CA Datacom Products  
/*  
//           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;  
/*
```

Following is sample z/VSE JCL.

**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.

```
* $$ JOB ...          See the note above..
* $$ 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,SIZE=AUTO
      SELECT * FROM table;
/*
/&
* $$ EOJ
```

## Example SQL Statements

The following SQL statements can be used to determine which plans on your system were added using the @TIMESTAMP keyword in the PLANAME option of your COBOL precompiler.

If you want to get a list of all plans on your system, that have been created using the new method of specifying @TIMESTAMP in the plan name of your COBOL program, you can code the following SQL to get the list output (AUTHID=SYSADM option specified):

```
SELECT PRIMARY_AUTH_ID, PRIMARY_OCC_NAM,
       HEX(SUBSTR(WORK_AREA_UPD, 81, 4)) AS SWITCHES
FROM DDD_TABLE
WHERE SQL_TYPE = 'PLN'
      AND BIT_AND(SUBSTR(WORK_AREA_UPD, 81, 1), X'02') = X'02'
ORDER BY PRIMARY_OCC_NAM
;
```

PRIMARY_AUTH_ID CHAR(32) NOT NULL	PRIMARY_OCC_NAM CHAR(32) NOT NULL	SWITCHES CHAR(8) .N.N
SYSADM	AVE2001_1507221647	02000000
SYSADM	AVE2001_1507231147	02000000
SYSADM	AVE2001_1507241436	02000000
SYSADM	AVE2001L1507231147	02000000
SYSADM	AVE2001L1507241436	02000000
SYSADM	AVE2001L1507241441	02000000
SYSADM	DAF4____1507221647	02000000
SYSADM	DAF4____1507231147	02000000
SYSADM	DAF4____1507241436	02000000
SYSADM	DAF888881507221647	02000000
SYSADM	DAF888881507231147	02000000
SYSADM	DAF888881507241436	02000000

\_\_\_ 12 rows returned \_\_\_

If you want a list of all your plans, with a column that indicates 'YES' if they were added by COBOL @TIMESTAMP specification, and 'NO' if they were not, you can use the following SQL statement:

```

SELECT PRIMARY_AUTH_ID, PRIMARY_OCC_NAM,
       (CASE
         WHEN (BIT_AND(SUBSTR(WORK_AREA_UPD, 81, 1), X'02') = X'02') THEN 'YES'
         ELSE 'NO'
        ) AS COBOL_TS
FROM DDD_TABLE
WHERE SQL_TYPE = 'PLN'
ORDER BY PRIMARY_OCC_NAM
;

```

PRIMARY_AUTH_ID CHAR(32) NOT NULL	PRIMARY_OCC_NAM CHAR(32) NOT NULL	COBOL_TS CHAR(3)
...		
SYSUSR	AVE1958	NO
SYSADM	AVE2001	NO
...		
SYSADM	AVE2001_1507221647	YES
SYSADM	AVE2001_1507231147	YES
SYSADM	AVE2001_1507241436	YES
...		
SYSADM	AVE2001L1507231147	YES
SYSADM	AVE2001L1507241436	YES
SYSADM	AVE2001L1507241441	YES
SYSADM	BRM1780	NO
...		
SYSADM	DAF4____1507221647	YES
SYSADM	DAF4____1507231147	YES
SYSADM	DAF4____1507241436	YES
...		
SYSADM	DAF888881507221647	YES
SYSADM	DAF888881507231147	YES
SYSADM	DAF888881507241436	YES
SYSADM	DICKTST	NO
SYSADM	DICKUDFT	NO
SYSADM	DICK309	NO
...		
___ nnn rows returned ___		



## Sample Report

Following are the first two pages of a DBSQLPR SQL Processor report. For a sample header for this report, see Sample Report Headers.

### Command Line Options

```
-----
INPUTWIDTH=72
OPTFILE=0P
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 CHAR(18)	COLINT INT
colChar row 1	1
colChar row 2	2

```
___ 2 rows returned ___
```

#### INPUT STATEMENT:

```
rollback work;
```

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

```
=====
== 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](#) (see page 705)), 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 141) 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:** 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.

## Datadictionary Considerations

No change is required by CA Datacom Datadictionary to provide additional data in the Data Definition Directory (DDD) table for SQL. All columns in a redefinition are already present and contain a unique SQL name, and all arrays are indicated with their number of elements.

## Datadictionary SQL Column Report

To aid users of a DATACOM VIEW, CA Datacom Datadictionary provides a new report to list the SQL columns that are visible to the view. See the *CA Datacom Datadictionary Batch Reference Guide*.

## Using SQC Table to Cancel SQL Requests

**Note:** For information about how to use the SQC Dynamic System Table to cancel SQL requests, see the CA Datacom/DB System Tables Reference Guide.

## Overriding SQL Key Selection

Although the SQL Optimizer normally automatically selects the most efficient key for you, occasionally the statistics used to estimate costs do not reflect actual costs closely enough to select the most efficient key. When this occurs, you can specify the key to be used in a special format of a correlation name, or if the SQL statement is generated and you cannot specify a correlation name, in a synonym name using the same special format.

Following is the syntax of the override key option:

►► `x_HINT_keynm` ————— ◀◀

**x**

is any value that is meaningful to you.

**Valid Entries:**

any leading characters

**Default Value:**

(No default)

**\_HINT\_**

is the literal that triggers the interpretation of the next five characters as the override key.

**Valid Entries:**

\_HINT\_

**Default Value:**

(No default)

**keynm**

is the override key, a 5-character internal key name, *not* the SQL key name. If this override key name exists for the table, all other keys are marked "ignore" in the Compound Boolean Selection Optimization Report, and all other keys are hidden from the SQL Optimizer. Be aware, however, that if the specified key name does not exist or cannot be used because it does not index nil values, no error is generated and the key is ignored.

No check is made to qualify the key for performance.

That the key was used can be reported by Accounting Element CBSOR (CBS Optimizer Reasons) as key type "P," that is, Parm, because it is specified in the CBS RQA (Request Qualification Area) Parameter section as the override key. The SQL Optimization Report also indicates the use of the override key under an "OVERRIDE KEY" heading.

**Valid Entries:**

a valid 5-character internal key name (not the SQL name for the key)

**Default Value:**

(No default)



## Examples

```
CREATE TABLE CARS (MAKE CHAR(8), COLOR CHAR(8));
CREATE INDEX CARS_COLOR_KEY ON CARS (COLOR) DATACOM NAME COLOR;
CREATE SYNONYM CARS_HINT_COLOR FOR CARS;
```

**Note:** The first line in the previously shown example creates a key on column MAKE.

The following example shows the use of a synonym for override:

```
SELECT * FROM CARS_HINT_COLOR
WHERE MAKE = 'FORD' AND COLOR= 'PINK';
```

The following example shows the use of a correlation name for override:

```
SELECT * FROM CARS CARS_HINT_COLOR
WHERE MAKE = 'FORD' AND COLOR= 'PINK';
```

**Note:** Using statistics such as *cardinality* and *BLKCHG* (how closely does key sequence reflect the physical sequence of rows in the data area), the SQL Optimizer might select the key on column MAKE when the key on COLOR is the most efficient, since PINK is a rare color. These queries cause the COLOR key to be used.

## XML Support

Using XML (Extensible Markup Language) allows you to *externalize* relational data as XML. The CA Datacom/DB implementation of XML includes support for the following XML functions:

- XMLATTRIBUTES
- XMLCONCAT
- XMLELEMENT
- XMLFOREST
- XMLSERIALIZE

For details about using XML, see [XML Functions](#) (see page 574).

## SQL Read-Only

SQL read-only tables can be defined by specifying the SQL-INTENT attribute-type as R in CA Datacom Datadictionary batch or online.

In a table defined as SQL read-only, using SQL Data Definition Language (DDL) statements to attempt to insert new rows, update existing rows, or delete existing rows is not allowed. Attempts to do so result in an error message.

**Note:** For more information on SQL read-only tables, see the *CA Datacom/DB Database and System Administration Guide*, the *CA Datacom Datadictionary Batch Reference Guide*, and the *CA Datacom Datadictionary Attribute Reference Guide*.

# Chapter 4: CA Datacom/DB SQL Preprocessors

---

Before your source program is compiled, the SQL statements embedded in the host language program must be prepared (in a process called binding) by an CA Datacom/DB SQL Preprocessor for COBOL, PL/I, Assembler, or C (for the C language, the only supported compiler is IBM LE/370 C).

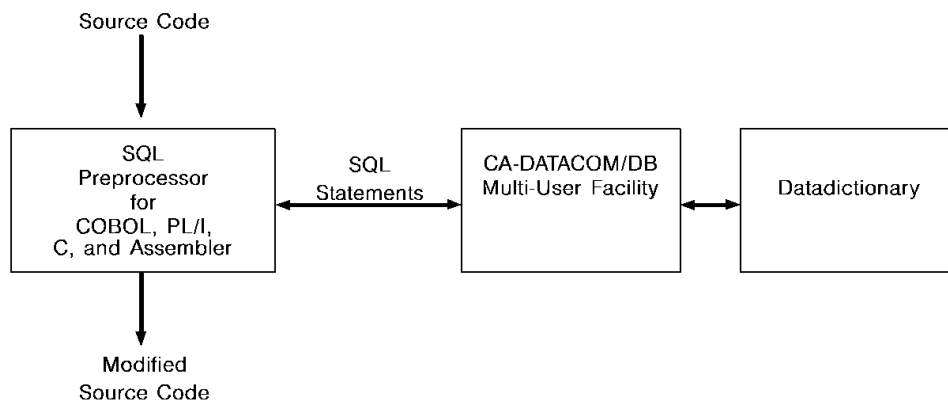
**Note:** Support for procedures and triggers requires a minimum of z/OS Version 2 Release 5 and compatible Language Environment for z/OS with the following IBM compiler products: z/OS C/C++ (only the C subset is supported), COBOL for z/OS, and PL/I for MVS.

## Overview

The CA Datacom/DB SQL Preprocessor prepares the data sublanguage portions of a source program for execution by:

- Scanning each statement in the program, and
- Producing a modified program in which every embedded SQL statement has been replaced by one or more statements of the host language.

The following diagram approximates how this preparation phase is handled in the CA Datacom/DB environment.



The Preprocessor:

- Recognizes SQL statements and passes them to CA Datacom/DB for compilation or processing into a CA Datacom/DB access plan.
- Replaces each executable SQL statement with source program calls to CA Datacom/DB to execute the previously compiled or processed statement.

The Preprocessor makes each embedded SQL statement in the host source into a comment and leaves it for documentation purposes. Each such executable statement is replaced with host language code to accomplish the required action.

- Identifies host-variables both where they are declared and where they are used in SQL statements, and creates appropriate references between them.
- Creates appropriate initialization code.

If errors are detected in SQL statements or supporting data definitions, the Preprocessor issues appropriate error messages and takes appropriate action.

The CA Datacom/DB access plan is the control structure produced during the bind process. This preparation phase builds the plan for the application and binds each SQL statement to the appropriate table, view or synonym definition stored in CA Datacom Datadictionary. CA Datacom/DB uses the plan to process SQL statements encountered during application execution. Each program containing embedded SQL statements must have a plan before being executed.

If any dependencies (tables, views and synonyms) for a prepared statement change, the related statement is marked invalid and must be rebound before it can be executed again.

The SQL Manager automatically attempts a rebound when an invalid statement is executed. Rebinding can also be requested in advance. For more information, see [Rebinding a Plan](#) (see page 457).

## Input to the Preprocessor

The primary input to the CA Datacom/DB Preprocessor consists of statements in the host language and embedded SQL statements. Both the host language and SQL statements must be written using the same margins. The input data set (SYSIN for COBOL, SOURCE for PL/I, Assembler, and C) must have attributes of fixed-length records, blocked or unblocked, and a record length of 80 bytes. If you are using C, however, note that the C Preprocessor only looks for data in columns 1 through 72. If you are using PL/I, Assembler, or C, the source data set for z/OS is either instream, physical sequential, or a member of a partition data set, or for z/VSE either a sequential disk file or a library member specified by the SMBR= Preprocessor option.

## INCLUDEs in COBOL

In COBOL, the SQL INCLUDE directive can be used to get secondary input from the include library, INCLUDE. The INCLUDE directive causes input to be read from the specified member of the include library until the end of the member is reached. The included library input cannot contain other Preprocessor INCLUDE directives, but can contain both host language and SQL statements. The include library must have fixed-length records of 80 bytes. For more information, see [INCLUDE Directive in COBOL](#) (see page 184).

## INCLUDEs in PL/I

In PL/I, the source input may have INCLUDE directives within certain restrictions. INCLUDE directives are of two types: PL/I and EXEC SQL. The PL/I INCLUDEs are standard for the language. The Preprocessor does not perform the functions of the PL/I preprocessor. In particular, it does not expand PL/I INCLUDE directives. To use code contained in PL/I INCLUDEs, you must run the Preprocessor to create source with the PL/I INCLUDEs expanded. The expanded source is the input to the Preprocessor. The Preprocessor does expand INCLUDEs designated within an EXEC SQL. The principal use here is for host declarations. The Preprocessor does not expand any nested INCLUDEs nor does it perform any preprocessor functions. INCLUDEs for the SQLCA, SQLWA or other SQL control blocks are ignored except to comment the lines in the source. SQL INCLUDEs must be in a single partition data set for z/OS and sublibraries of a single library for z/VSE. The include data set is INCLUDE. For more information, see [Rules for SQL INCLUDEs in PL/I](#) (see page 194).

## INCLUDEs in Assembler

In Assembler, the Preprocessor (except for SQL includes) does not expand macros or include source. Macros, copy statements, or other includes are ignored during preprocessing for SQL use. They are, however, written to the modified source for processing by the Assembler. For more information, see [Rules for SQL INCLUDEs in Assembler](#) (see page 203).

## INCLUDEs in C

INCLUDEs in the C language are basically the same as in PL/I except note the following:

- The SQLCA is always automatically generated, so no INCLUDE should be used for it. However, you must code your own SQLDA and SQLVAR when using dynamic SQL.
- The EXEC SQL INCLUDE <member name>; SQL statement can be used to include members from the PDS data set INCLUDE in z/OS (for z/VSE, the name in the EXEC SQL INCLUDE is the name of a member in a z/VSE library). INCLUDEs in C cannot be nested.
- Because the Precompiler executes before the C Preprocessor, statements such as #include, #define, and typedef are not expanded. Unless the user executes the language preprocessor only before the CA Datacom/SQL C Preprocessor, host variables in #include files cannot be referenced, and #define and typedefs cannot be used to declare a host variable data type.

## Output from the Preprocessors

### COBOL

The various kinds of output from the SQL Preprocessor for COBOL are:

#### **Listing Output**

The following listings are written on the output data set.

##### **Preprocessor Source Listing**

This listing shows Preprocessor source statements with line numbers assigned by the Preprocessor.

##### **Preprocessor Diagnostics**

Diagnostic messages follow immediately after the statement(s) in error.

#### **Translated Source Statements**

Source statements processed by the Preprocessor are written to SYSPUNCH (z/OS environment) or SYSPCH (z/VSE environment), the data set designated as input to the compiler. This data set must have attributes of fixed-length records, blocked or unblocked, and a record length of 80 bytes. In your modified source code, SQL statements have been converted to comments and calls to the SQL Manager.

#### **Work Data Sets**

Three data sets by the symbolic names &.&WORK1., &.&WORK2., and &.&WORK3., are required by the Preprocessor. These data sets must have attributes of fixed-length records, blocked or unblocked, and a record length of 80 bytes. These work areas must be large enough to hold the generated output.

## PL/I, Assembler, and C

The various kinds of output from the SQL Preprocessor for PL/I, Assembler, and the C language are:

### **Listing Output**

The following sections of the report are written to REPORT.

#### **Options File**

This section of the report shows record images with diagnostics.

#### **Execution Parameters**

This section of the report shows the added execution parameters with diagnostics.

#### **Preprocessor Source Listing with Diagnostics**

This section of the report shows the unmodified source statements with line numbers assigned by the Preprocessor. Diagnostic messages follow immediately after the statement(s) in error.

#### **Preprocessor-modified Source Listing**

This section of the report shows the source code modified for input.

### **Translated Source Statements**

Source statements processed by the Preprocessor are written to SRCOUT, the data set designated as input to the compiler. This data set must have attributes of fixed-length records, blocked or unblocked, and a record length of 80 bytes. In your modified source code, SQL statements have been converted to comments and calls to the SQL Manager.

## Sample JCL

If you are coding a procedure, see Examples: Creating a Procedure.

Sample JCL for COBOL follows. Also see:

- [Sample PL/I JCL](#) (see page 161),
- [Sample Assembler JCL](#) (see page 163),
- [Sample C Language JCL](#) (see page 168).

## Sample COBOL JCL

The following examples show sample JCL for z/OS and z/VSE environments. These examples are intended to be samples only. You need to modify the JCL to conform to your site's requirements.

**Note:** If you are coding a procedure, see [Examples: Creating a Procedure](#) (see page 87).

### z/OS Sample COBOL JCL for Batch

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 and Listing Libraries for CA Datacom Products.
//*****
//*  THE FOLLOWING JOB STREAM DEMONSTRATES THE SQL
//*  PREPROCESSOR, COBOL COMPILER, AND PROGRAM EXECUTION
//*****
//STEP1    EXEC PGM=DBXMMPR
//STEPLIB  See the note above and Listing Libraries for CA Datacom Products.
//WORK1    DD  DSN=&. &WORK1 . ,UNIT=SYSDA,DISP=(NEW,PASS),
//          DCB=(RECFM=F,LRECL=80,BLKSIZE=80),SPACE=(TRK,(1,1))
//WORK2    DD  DSN=&. &WORK2 . ,UNIT=SYSDA,DISP=(NEW,PASS),
//          DCB=(RECFM=F,LRECL=80,BLKSIZE=80),SPACE=(TRK,(1,1))
//WORK3    DD  DSN=&. &WORK3 . ,UNIT=SYSDA,DISP=(NEW,PASS),
//          DCB=(RECFM=F,LRECL=80,BLKSIZE=80),SPACE=(TRK,(1,1))
//SYSOUT   DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*                               Print Output
//SYSPUNCH DD  DSN=&. &TEMP . ,UNIT=SYSDA,DISP=(NEW,PASS),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=800),SPACE=(TRK,(2,1))
//SYSUDUMP DD  SYSOUT=*
//SNAPER   DD  SYSOUT=*
//INCLUDE  DD  DSN=ca.user.include.library,DISP=SHR
//SYSIN    DD  *                                       Command input
```



```

                PLACE COBOL SOURCE TEXT HERE.
//*****
//*          COBOL COMPILE
//*****
//STEP2 EXEC COBUC,COND=(0,NE,STEP1),
//      PARM.COBS='LIST,NODYNAM,SXREF,PMAP,DMAP'
//COB.SYSPRINT DD SYSOUT=*
//COB.SYSLIN  DD DSN=&. &DCMPUNCH.,DISP=(NEW,PASS,DELETE),
//          UNIT=VIO,SPACE=(TRK,(15,15)),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//COB.SYSIN  DD DSN=&. &TEMP.,UNIT=SYSDA,DISP=(OLD,DELETE,DELETE),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
/*
//*****
//*          LINK EDIT
//*****
//LINKSTP EXEC PGM=IEWL,PARM='LIST,XREF,LET',COND=(0,NE)
//SYSPRINT  DD SYSOUT=*
//SYSLMOD   DD DSN=ca.user.loadlib,DISP=SHR
//SYSUT1    DD UNIT=VIO,SPACE=(1024,(200,20))
//SYSLIB    DD DSN=ca.cobol.compiler.loadlib,DISP=SHR
//OBJLIB    DD DSN=ca.user.smp.library,DISP=SHR
//SYSLIN    DD DSN=&. &DCMPUNCH.,DISP=(OLD,PASS)
//          DD *
        INCLUDE OBJLIB(DBXHVPR)
        INCLUDE OBJLIB(DBSBTPR) (Use DBSU1PR if program is AMODE=31 and RMODE=ANY.)
        ENTRY BEGIN
        NAME DBDP236(R)

/*
//*****
//*          PROGRAM EXECUTION
//*****
//STEP4 EXEC PGM=DBDP236
//STEPLIB DD DSN=ca.datacom.loadlib,DISP=SHR See Listing Libraries for CA Datacom
Products.
//          DD DSN=ca.user.loadlib,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIST  DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//PRINT    DD SYSOUT=*
/*

```

## z/OS Sample COBOL JCL for CICS

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 and Listing Libraries for CA Datacom Products.
//*****
//* THE FOLLOWING JOB STREAM DEMONSTRATES THE SQL
//* PREPROCESSOR, THE CICS COMMAND LEVEL PRECOMPILER AND THE COBOL
//* COMPILER STEPS
//*****
//STEP1     EXEC PGM=DBXMMPR
//STEPLIB   See the note above and Listing Libraries for CA Datacom Products.
//WORK1     DD DSN=&. &WORK1. ,UNIT=SYSDA,DISP=(NEW,PASS),
//           DCB=(RECFM=F,LRECL=80,BLKSIZE=80),SPACE=(TRK,(1,1))
//WORK2     DD DSN=&. &WORK2. ,UNIT=SYSDA,DISP=(NEW,PASS),
//           DCB=(RECFM=F,LRECL=80,BLKSIZE=80),SPACE=(TRK,(1,1))
//WORK3     DD DSN=&. &WORK3. ,UNIT=SYSDA,DISP=(NEW,PASS),
//           DCB=(RECFM=F,LRECL=80,BLKSIZE=80),SPACE=(TRK,(1,1))
//SYSOUT    DD SYSOUT=*
//SYSPRINT  DD SYSOUT=*                               Print Output
//SYSPUNCH  DD DSN=&. &TEMP. ,UNIT=SYSDA,DISP=(NEW,PASS),
//           DCB=(RECFM=FB,LRECL=80,BLKSIZE=800),SPACE=(TRK,(2,1))
//SYSUDUMP  DD SYSOUT=*
//SNAPER    DD SYSOUT=*
//INCLUDE   DD DSN=ca.user.include.library,DISP=SHR
//SYSIN     DD *                                       Command input
           PLACE COBOL SOURCE TEXT HERE.
//*****
//* CICS COMMAND LEVEL PREPROCESSOR STEP
//*****
//TRN       EXEC PGM=DFHECP1$,COND=(4,GT)
//STEPLIB   DD DSN=cics.loadlib,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSPUNCH  DD DSN=&. &SYSCIN. ,DISP=(,PASS),UNIT=DISK,
//           DCB=BLKSIZE=400,SPACE=(400,(400,100))
//SYSIN     DD DSN=&. &TEMP. ,UNIT=DISK,DISP=(OLD,DELETE,DELETE),
//           DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
//SYSLIN    DD DSN=&. &DCMPUNCH. ,DISP=(NEW,PASS,DELETE),
//           UNIT=VIO,SPACE=(TRK,(15,15)),
//           DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
```

```

//*****
//*  COBOL COMPILER STEP
//*****
//COB      EXEC PGM=IKFCBL00,REGION=1024K,
//          PARM='NOTRUNC,NODYNAM,LIB,SIZE=1024K,BUF=16K',
//          COND=(4,GT)
//SYSLIB   DD DSN=cics.coblib,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSIN    DD DSN=&.&SYSCIN.,DISP=(OLD,DELETE)
//SYSLIN   DD DSN=&.&LOADSET.,DISP=(MOD,PASS),
//          UNIT=DISK,SPACE=(80,(250,100))
//SYSUT1   DD UNIT=DISK,SPACE=(460,(350,100))
//SYSUT2   DD UNIT=DISK,SPACE=(460,(350,100))
//SYSUT3   DD UNIT=DISK,SPACE=(460,(350,100))

//*****
//*  LINK EDIT STEP
//*****
//LKED     EXEC PGM=IEWL,REGION=1024K,PARM=XREF,COND=(4,GT)
//SYSLIB   DD DSN=cics.loadlib,DISP=SHR
//          DD DSN=SYS1.COBLIB,DISP=SHR
//          DD DSN=ca.datacom.loadlib,DISP=SHR See Listing Libraries for CA Datacom
Products.
//SYSLMOD  DD DSN=DATACOM.LOADLIB,DISP=SHR
//SYSUT1   DD UNIT=DISK,DCB=BLKSIZE=1024,
//          SPACE=(1024,(200,20))
//SYSPRINT DD SYSOUT=*
//SYSLIN   DD DSN=cics.coblib(DFHEILIC),DISP=SHR
//          DD DSN=&.&LOADSET.,DISP=(OLD,DELETE)
//SYSIN    DD *
          INCLUDE SYSLIB(DBCSVPR)
          INCLUDE SYSLIB(DBXHVPR)
          NAME TEST01(R)
/*
//

```

## CA Datacom IMS/DC Services Sample COBOL z/OS JCL

The following z/OS JCL example is for compilation of programs running under CA Datacom IMS/DC Services.

**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 and Listing Libraries for CA Datacom Products.
//*****
//*   THE FOLLOWING JOB STREAM DEMONSTRATES THE SQL
//*   PREPROCESSOR, COBOL COMPILER, AND PROGRAM EXECUTION
//*****
//STEP1 EXEC PGM=DBXMMPR
//STEPLIB   See the note above and Listing Libraries for CA Datacom Products.
//WORK1 DD   DSN=&.&WORK1.,UNIT=SYSDA,DISP=(NEW,PASS),
//          DCB=(RECFM=F,LRECL=80,BLKSIZE=80),SPACE=(TRK,(1,1))
//WORK2 DD   DSN=&.&WORK2.,UNIT=SYSDA,DISP=(NEW,PASS),
//          DCB=(RECFM=F,LRECL=80,BLKSIZE=80),SPACE=(TRK,(1,1))
//WORK3 DD   DSN=&.&WORK3.,UNIT=SYSDA,DISP=(NEW,PASS),
//          DCB=(RECFM=F,LRECL=80,BLKSIZE=80),SPACE=(TRK,(1,1))
//SYSOUT DD   SYSOUT=*
//SYSPRINT DD  SYSOUT=*                               Print Output
//SYSPUNCH DD  DSN=&.&TEMP.,UNIT=SYSDA,DISP=(NEW,PASS),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=800),SPACE=(TRK,(2,1))
//SYSUDUMP DD  SYSOUT=*
//SNAPER DD   SYSOUT=*
//INCLUDE DD  DSN=ca.user.include.library,DISP=SHR
//SYSIN DD   *                                       Command input

          PLACE COBOL SOURCE TEXT HERE.
//*****
//*   COBOL COMPILE
//*****

//STEP2 EXEC COBUC,COND=(0,NE,STEP1),
//          PARM.COB='LIST,NODYNAM,SXREF,PMAP,DMAP'
//COB.SYSPRINT DD  SYSOUT=*
//COB.SYSLIN DD  DSN=&.&DCMPUNCH.,DISP=(NEW,PASS,DELETE),
//          UNIT=VIO,SPACE=(TRK,(15,15)),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//COB.SYSIN DD  DSN=&.&TEMP.,UNIT=SYSDA,DISP=(OLD,DELETE,DELETE),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
/*
```

```

//*****
//*      LINK EDIT
//*****
//LINKSTP EXEC PGM=IEWL,PARM='LIST,XREF,LET',COND=(0,NE)
//SYSPRINT DD SYSOUT=*
//SYSLMOD DD DSN=ca.user.loadlib,DISP=SHR
//SYSUT1 DD UNIT=VIO,SPACE=(1024,(200,20))
//SYSLIB DD DSN=ca.cobol.compiler.loadlib,DISP=SHR
// DD DSN=ca.datacom.loadlib,DISP=SHR See Listing Libraries for CA Datacom
Products.
//IMSDCLIB DD DSN=yourimsdclib,DISP=SHR
//SYSLIN DD DSN=&. &DCMPUNCH.,DISP=(OLD,PASS)
// DD *
INCLUDE SYSLIB(DBXHVR)
INCLUDE IMSDCLIB(CBLTDLI)
ENTRY DLITCBL
NAME PROGNAME(R)
//*****
//* NOTE: IMSDCLIB IS THE LIBRARY CONTAINING THE
//* CA-DATACOM/IMSDC SERVICES LANGUAGE INTERFACE.
//*****
/*

```

## z/VSE Sample COBOL JCL for Batch

The following sample JCL is for z/VSE 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.

```

* $$ JOB ... See the note above and Listing Libraries for CA Datacom Products.
* $$ LST CLASS=x
// JOB name
// EXEC PROC=yourproc Whether you use PROCs or LIBDEFs, see Listing Libraries
for CA Datacom Products.
// 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 xxxxxxxx, *
      INCLUDE DBSBTPR
      INCLUDE DBXHVPR
      ASSGN   SYSIPT, DISK, VOL=volser, SHR
// EXEC IGYCRCTL, SIZE=IGYCRCTL
/*
      CLOSE   SYSIPT, READER
/*
* LNKEDT
      ENTRY   BEGIN
// EXEC LNKEDT
/*
// EXEC
```

```

/*
/ &
* $$ E0J

```

## z/VSE Sample COBOL JCL for CICS

The following sample JCL is for z/VSE 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.

```

* $$ JOB ...          See the note above and Listing Libraries for CA Datacom Products.
* $$ LST  CLASS=x
// JOB    name
// EXEC   PROC=yourproc
// 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
* PRECOMPILE
// OPTION DECK,NOXREF,DUMP,LOG
// EXEC DBXMMPR,SIZE=768K
CBL LIB,APOST
      cobol source
/*
* DFHECP1$
// DLBL   IJSYSPH,'source.file',0,SD
// EXTENT SYSPCH,volser,1,0,reltrk,numtrks
      ASSGN SYSPCH,DISK,VOL=volser,SHR
// DLBL   IJSYSIN,'source.name'
// EXTENT SYSIPT,volser
      ASSGN SYSIPT,DISK,VOL=volser,SHR
// EXEC DFHECP1$,SIZE=512K
/*
CLOSE   SYSPCH,PUNCH
CLOSE   SYSIPT,READER

```

```
* COMPILE
// DLBL      IJSYSIN, 'source.file'
// 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
// OPTION   NODECK,CATAL
//          PHASE xxxxxxxx,*
//          INCLUDE DBSBTPR
//          INCLUDE DBXHVPR
// ASSGN    SYSIPT,DISK,VOL=volser,SHR
// EXEC IGYCRCTL,SIZE=IGYCRCTL
/*
// CLOSE    SYSIPT,READER
/*
* LNKEDT
// ENTRY   BEGIN
// EXEC LNKEDT
/*
/&
* $$ EOJ
```



## Sample PL/I JCL

The following examples show sample JCL for z/OS and z/VSE environments. These examples are intended to be samples only. You need to modify the JCL to conform to the requirements of your site.

**Note:** If you are coding a procedure, see [Examples: Creating a Procedure](#) (see page 87).

### z/OS PL/I Sample JCL

**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 and Listing Libraries for CA Datacom Products.
/*-----
/* THE FOLLOWING JOB STREAM ILLUSTRATES THE EXECUTION OF THE
/* PREPROCESSOR AND COMPILER.
/*-----
/*
/*-----
/* PRECOMPILE
/*-----
/*
//STEP1 EXEC PGM=DBPLIPR
//STEPLIB    See the note above and Listing Libraries for CA Datacom Products.
//SYSUDUMP DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSOUT    DD SYSOUT=*
//OPTIONS   DD *,DCB=BLKSIZE=80
LANGUAGE=PLI
... place additional precompile options here ....
/*
```

```
//SOURCE DD DATA,DCB=BLKSIZE=80,DLM=##
... place PL/I source here ...
##
//INCLUDE DD DSN=ca.user.include.library,DISP=SHR
//SRCOUT DD DSN=&.&SRCPRE.,DISP=(MOD,PASS),UNIT=SYSDA,
//          DCB=(LRECL=80,BLKSIZE=4000,RECFM=FB),
//          SPACE=(4000,(250,100))
//REPORT DD SYSOUT=*
//*
/*-----
/* assemble urt
/*-----
/*
//ASMURT EXEC PGM=ASMA90,PARM='DECK,NOOBJECT,TERM'
//SYSLIB DD DSN=ca.user.system.library,DISP=SHR
//SYSUT1 DD DSN=&.&SYSUT1.,UNIT=VIO,SPACE=(1700,(600,100))
//SYSUT2 DD DSN=&.&SYSUT2.,UNIT=VIO,SPACE=(1700,(300,50))
//SYSUT3 DD DSN=&.&SYSUT3.,UNIT=VIO,SPACE=(1700,(300,50))
//SYSTEM DD SYSOUT=*,DCB=BLKSIZE=1089
//SYSPUNCH DD DSN=&.&URTPUNCH.,DISP=(NEW,PASS,DELETE),
//          UNIT=VIO,SPACE=(3200,(15,15)),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSIN DD *
... place urt source here ....
... urt is opened by CA-DATACOM/DB ...
/*

//SYSPRINT DD SYSOUT=*
/*
/*-----
/*          COMPILER & LINK
/*-----
/*
//CC EXEC PLIXCL
//PLI.SYSLIN DD UNIT=SYSDA
//PLI.SYSIN DD DSN=&.&SRCPRE.,DISP=(MOD,PASS),UNIT=SYSDA,
//          DCB=(LRECL=80,BLKSIZE=4000,RECFM=FB)
//LKED.SYSLMOD DD DSN=ca.user.loadlib,DISP=SHR
//LKED.SYSLIN DD DSN=&.&URTPUNCH.,DISP=(OLD,DELETE)
//          DD DDNAME=SYSIN
//LKED.SYSIN DD *
ENTRY BEGIN
NAME TESTPL1(R)
/*
//
```

## z/VSE PL/I Sample JCL

In this example, both the source and the SQL INCLUDEs are being read from libraries. The desired source is specified by the SMBR= Preprocessor option. A library location is required when specifying SMBR=. In addition, the file type is specified by using the ITYP= Preprocessor option. For more information, see Description of Options.

**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.

```
* $$ JOB ...      See the note above and Listing Libraries for CA Datacom Products.
* $$ LST ...
* $$ PUN ...
// JOB name
// DLBL CAC,'name.c.runtime'
// EXTENT ,DDSRN3
// DLBL INCLUDE,'name.sql.includes',99/365
// EXTENT ,volser...
// DLBL SOURCE,'name.pli.source',99/365
// EXTENT ,volser...
// DLBL DATACOM,'name.datacom.db'
// EXTENT ,volser...
// LIBDEF *,SEARCH=(DATACOM.LINK2480,CAC.CORE,
                    SOURCE.PROD,INCLUDE.PLISQL )
// DLBL OPTIONS,'name.precomp.pli.options',0,SD
// EXTENT SYSnnn,volser...
// DLBL REPORT,'name.precomp.pli.report',0,SD
// EXTENT SYSnnn,volser...
// DLBL SRCOUT,'name.precomp.pliout',0,SD
// EXTENT SYSnnn,volser...
// ASSGN SYSnnn,DISK,VOL=volser,SHR
// ASSGN SYSnnn,DISK,VOL=volser,SHR
// EXEC PGM=DBPLIPR,SIZE=500K,PARM=' SMBR=PLISRC4.P ITYP=S'
/*
/&
* $$ E0J
```

## Sample Assembler JCL

The following examples show sample JCL for z/OS and z/VSE environments. These examples are intended to be samples only. You need to modify the JCL to conform to the requirements of your site. Depending on the method you use, some steps may be omitted.

**Note:** If you are coding a procedure, see [Examples: Creating a Procedure](#) (see page 87).

## z/OS Assembler Sample JCL

**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 and Listing Libraries for CA Datacom Products.
/*-----
/* the following job stream illustrates this sequence:
/* preprocess, assemble, link, & execute
/*-----
/*
/*-----
/* preprocess program
/*-----
/*
//PREC EXEC PGM=DBPLIPR
//STEPLIB   See the note above and Listing Libraries for CA Datacom Products.
//SYSUDUMP DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//OPTIONS  DD *,DCB=BLKSIZE=80
LANGUAGE=ASM
... place additional preprocessor options here ....
/*
```

```

//SOURCE DD DATA,DCB=BLKSIZE=80,DLM=##
... place source here ...
##
//SRCOUT DD DSN=&.&SRCPRE.,DISP=(MOD,PASS),UNIT=SYSDA,
//      DCB=(LRECL=80,BLKSIZE=4000,RECFM=FB),
//      SPACE=(80,(250,100))
//REPORT DD SYSOUT=*
//INCLUDE DD DSN=ca.user.include.library,DISP=SHR
//*
/*-----
/* assemble program
/*-----

/*
//ASMURT EXEC PGM=ASMA90,PARM='DECK,NOOBJECT,TERM'
//SYSLIB DD DSN=ca.user.system.library,DISP=SHR
//SYSUT1 DD DSN=&.&SYSUT1.,UNIT=VIO,SPACE=(1700,(600,100))
//SYSUT2 DD DSN=&.&SYSUT2.,UNIT=VIO,SPACE=(1700,(300,50))
//SYSUT3 DD DSN=&.&SYSUT3.,UNIT=VIO,SPACE=(1700,(300,50))
//SYSTEM DD SYSOUT=*,DCB=BLKSIZE=1089
//SYSPUNCH DD DSN=&.&ASPUNCH.,DISP=(NEW,PASS,DELETE),
//      UNIT=VIO,SPACE=(3200,(15,15)),
//      DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSIN DD DSN=&.&SRCPRE.,DISP=(OLD,PASS),UNIT=SYSDA,
//      DCB=(LRECL=80,BLKSIZE=4000,RECFM=FB)
//SYSPRINT DD SYSOUT=*
/*
/*-----
/* assemble urt
/*-----
/*
//ASMURT EXEC PGM=ASMA90,PARM='DECK,NOOBJECT,TERM'
//SYSLIB DD DSN=ca.user.system.library,DISP=SHR
//      DD DSN=ca.user.system.library,DISP=SHR
/*      DD DSN=ca.user.system.library,DISP=SHR
//SYSUT1 DD DSN=&.&SYSUT1.,UNIT=VIO,SPACE=(1700,(600,100))
//SYSUT2 DD DSN=&.&SYSUT2.,UNIT=VIO,SPACE=(1700,(300,50))
//SYSUT3 DD DSN=&.&SYSUT3.,UNIT=VIO,SPACE=(1700,(300,50))
//SYSTEM DD SYSOUT=*,DCB=BLKSIZE=1089
//SYSPUNCH DD DSN=&.&URTPUNCH.,DISP=(NEW,PASS,DELETE),
//      UNIT=VIO,SPACE=(3200,(15,15)),
//      DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSIN DD *
... place urt source here ...
... urt is opened by CA-DATACOM/DB ...
/*

```

```
//SYSPRINT DD SYSOUT=*
//*
/*-----
/* link program
/*-----
/*
//LINK EXEC PGM=IEWL,REGION=512K,
// COND=(8,LT),PARM='XREF,LIST,MAP,NCAL,LET'
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSN=&.&ASMPUNCH.,DISP=(OLD,DELETE)
// DD DSN=&.&URTPUNCH.,DISP=(OLD,DELETE)
// DD DDNAME=SYSIN
//SYSLMOD DD DSN=ca.user.loadlib,DISP=SHR
//SYSUT1 DD UNIT=VIO,SPACE=(1024,(400,40))
//LINK.SYSIN DD *
ENTRY BEGIN
NAME TESTASM(R)
/*
/*
/*-----
/* exec program
/*-----
/*
//ASMEXEC EXEC PGM=TESTASM,REGION=512K
//STEPLIB See note at start of this example and Listing Libraries for CA Datacom
Products.
//SYSUDUMP DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSOUT DD SYSOUT=*
//SNAP DD SYSOUT=*
```

## z/VSE Assembler Sample JCL

In this example, the source is being read from a sequential disk file while SQL INCLUDEs are being read from a library.

**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.

```
* $$ JOB .....
// JOB .....
// EXEC PROC=procname
// ASSGN SYSnnn,DISK,VOL=vvvvvv,SHR
// DLBL OPTIONS,'optdsn',1
// EXTENT SYSnnn,vvvvvv,1,0,strk,ntrks
// DLBL SOURCE,'srcdsn',1
// EXTENT SYSnnn,vvvvvv,1,0,strk,ntrks
// DLBL REPORT,'rptdsn',1
// EXTENT SYSnnn,vvvvvv,1,0,strk,ntrks
// DLBL SRCOUT,'outdsn',1
// EXTENT SYSnnn,vvvvvv,1,0,strk,ntrks
// DLBL SDSKIN,'rptdsn'
// EXTENT SYSnnn,vvvvvv
// UPSI 1
* DITTO
// EXEC DITTO,SIZE=1M
$$DITTO CSQ FILEOUT=OPTIONS,CISIZE=512,BLKFACTOR=1
SQLMODE=DATACOM
*** Other options as appropriate ***
/*
$$DITTO CSQ FILEOUT=SOURCE,CISIZE=512,BLKFACTOR=1
*** Source here ***
/*
$$DITTO EOJ
ON $ABEND GOTO $EOJ
* DBPLIPR
// EXEC DBPLIPR,SIZE=500K
/*
* DITTO
// EXEC DITTO,SIZE=1M
$$DITTO SFD FILEIN=SDSKIN,RECSIZE=133
$$DITTO EOJ
/*
/&
* $$ EOJ
```

## Sample C Language JCL

This is a sample only. Modify the JCL to conform to the requirements of your site. Depending on the method you use, some steps may be omitted.

**Note:** If you are coding a procedure, see [Examples: Creating a Procedure](#) (see page 87).

### z/OS C Language Sample JCL

**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 and Listing Libraries for CA Datacom Products.
/* -----
/* the following job stream illustrates the execution of the
/* preprocessor and compiler
/* -----
/*
/* -----
/*          c preprocessor
/* -----
/*
//CPRECOMP  EXEC PGM=progname,PARM='PLANNAME=TESTCEE'
//STEPLIB   See the note above and Listing Libraries for CA Datacom Products.
//SYSUDUMP  DD SYSOUT=*
//SYSPRINT  DD SYSOUT=*
//SYSOUT    DD SYSOUT=*
//OPTIONS   DD DATA,DCB=BLKISIZE=80,DLM=##
LANGUAGE=C
/* -----
/* place additional precompile options here
/* -----
//SOURCE    DD DATA,DCB=BLKISIZE=80,DLM=##
/* -----
/* place c source here
/* -----
##
//INCLUDE   DD DSN=ca.user.include.library,DISP=SHR
//REPORT    DD SYSOUT=*
//SRCOUT    DD DSN=&.&SRC.,DISP=(,PASS,DELETE),UNIT=VIO,
//          SPACE=(2000,(200,200)),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
/* -----
/*          c compile
/* -----
/*
```



```

//COMPA EXEC PROC=EDCC,
// CRUN='RENT',
// CPARAM='NOMARGINS,NOSEQUENCE,LIST,SOURCE',
// CPARAM2='LOCALE("POSIX")',
// CPARAM3='SSCOM, LONGNAME, SHOWINC, OMVS, DLL',
// INFILE='&.&SRC'. ,
// OUTFILE='ca.user.objlib(TESTCOBJ)',
// COND=(0,LT)
/*
//SYSLIB DD DSN=CEE.SCEEH.H,DISP=SHR
// DD DSN=CEE.SCEEH.SYS.H,DISP=SHR
//USERLIB DD DSN=ca.user.srclib,DISP=SHR
/* -----
/* prelink starts here
/* -----
//PRELINK EXEC PGM=EDCPRLK,COND=(0,LT),
// PARM='POSIX(OFF)/OE, MEMORY, DUP, NOER, MAP, NOUPCASE, NONCAL'
/*
//SYMSGS DD DSN=CEE.SCEEMSGP(EDCPMSGE),DISP=SHR
//OBJLIB DD DSN=ca.user.objlib,DISP=SHR
//C8941 DD DSN=CEE.SCEEOBJ,DISP=SHR
//SYSOUT DD SYSOUT=*
//SYSPRINT DD *
//SYSMOD DD DSN=ca.user.objlib(TESTCEE),DISP=SHR
//SYSDEFSD DD DUMMY
//SYSIN DD *
INCLUDE OBJLIB(TESTCOBJ)
LIBRARY C8941
/*
/* -----
/* link
/* -----
//LINKEDIT EXEC PGM=LINKEDIT,COND=(0,LT),
// PARM=('AMODE=31,RMODE=ANY,TERM=YES,MSGLEVEL=0,MAP,DYNAM=DLL',
// 'CALL=YES,CASE=MIXED,REUS=RENT,EDIT=YES')
//SYSLIB DD DSN=CEE.SCEELKED,DISP=SHR
// DD DSN=SYS1.CSSLIB,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTEM DD SYSOUT=*
//SYSLMOD DD DISP=SHR,DSN=ca.user.loadlib
//OBJLIB DD DSN=ca.user.objlib,DISP=SHR
//SYSLIN DD *
INCLUDE OBJLIB(urtcee)
INCLUDE OBJLIB(TESTCEE)
ENTRY BEGIN
NAME TESTCEE(R)
/*

```

## z/VSE C Language Sample JCL

**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.

```

* $$ JOB .....
// JOB .....
// EXEC PROC=procname
// LIBDEF *,CATALOG=lib.sublib
// ASSGN SYSnnn,DISK,VOL=nnnnnn,SHR
// DLBL OPTIONS,'optdsn',1
// EXTENT SYSnnn,vvvvvv,1,0,stk,ntrks
// DLBL SOURCE,'Srcdsn',1
// EXTENT SYSnnn,vvvvvv,1,0,stk,ntrks
// DLBL IJSYSPH,'pundsn',1
// EXTENT SYSnnn,vvvvvv,1,0,stk,ntrks
// DLBL IJSYSIN,'pundsn'
// EXTENT SYSIPT,vvvvvv
* ASSEMBLY
  ASSGN SYSPCH,DISK,VOL=vvvvvv,SHR
// OPTION NOXREF,DECK,NOEDECK,LOG,NODUMP ,CATAL
// EXEC ASSEMBLY
  *** urt assembly source ***
/*
* LIBR
  CLOSE SYSPCH,PUNCH
  ASSGN SYSIPT,DISK,VOL=vvvvvv,SHR
// EXEC LIBR,SIZE=450K,PARM='MSHP;ACC S=lib.sublib;
                                CATALOG urtname.OBJ,REPLACE=YES'
  CLOSE SYSIPT,YSRDR
* PUNCH PRECOMPILE OPTIONS AND SOURCE FILE
// UPSI 1
// EXEC DITTO,SIZE=1M
$$DITTO CSQ FILEOUT=OPTIONS,CISIZE=512,BLKFACTOR=1
SQLMODE=DATACOM
LANG=C
  *** Other options as appropriate
/*
$$DITTO CSQ FILEOUT=SOURCE,CISIZE=512,BLKFACTOR=1
  *** Source here ***
/*
$$DITTO EOJ
ON $ABEND GOTO $EOJ
// ASSGN SYSnnn,DISK,VOL=vvvvvv,SHR
// DLBL SOURCE,'srcdsn'
// EXTENT SYSnnn,vvvvvv
// DLBL OPTIONS,'optdsn'
// EXTENT SYSnnn,vvvvvv
// DLBL REPORT,'rptdsn',1
// EXTENT SYSnnn,vvvvvv,1,0,stk,ntrks
// DLBL SRCOUT,'srodsn',1
// EXTENT SYSnnn,vvvvvv,1,0,stk,ntrks
// DLBL IJSYS01,'work1',1,SD
// EXTENT SYS001,vvvvvv,1,0,stk,ntrks

```

```
// DLBL    IJSYS02, 'work2', 1, SD
// EXTENT  SYS001, vvvvvv, 1, 0, strk, ntrks
// DLBL    IJSYS03, 'work3', 1, SD
// EXTENT  SYS001, vvvvvv, 1, 0, strk, ntrks
// DLBL    IJSYS04, 'work4', 1, SD
// EXTENT  SYS001, vvvvvv, 1, 0, strk, ntrks
* PRECOMPILER STEP
// EXEC DBPLIPR, SIZE=500K
/*
// ASSGN   SYSnnn, DISK, VOL=vvvvvv, SHR
// DLBL    SDSKIN, 'rptdsn'
// EXTENT  SYSnnn, vvvvvv
* PRINT PRECOMPILER REPORT
// EXEC   DITTO, SIZE=1M
$$DITTO SFD FILEIN=SDSKIN, RECSIZE=133
$$DITTO EOJ
/*
* COMPILER STEP
// UPSI 0
// LIBDEF *, SEARCH=(cuslib, cussub, lib.csublib, lib.lesublib)
// DLBL   IJSYSLN, 'syslink', 0, SD
// EXTENT SYSLNK, vvvvvv, 1, 0, strk, ntrks
// DLBL   IJSYS01, 'work1', 1, SD
// EXTENT SYS001, vvvvvv, 1, 0, strk, ntrks
// DLBL   IJSYS02, 'work2', 1, SD
// EXTENT SYS002, vvvvvv, 1, 0, strk, ntrks
// DLBL   IJSYS03, 'work3', 1, SD
// EXTENT SYS003, vvvvvv, 1, 0, strk, ntrks
// DLBL   IJSYS04, 'work4', 1, SD
// EXTENT SYS004, vvvvvv, 1, 0, strk, ntrks
// ASSGN  SYS005, DISK, VOL=vvvvvv, SHR
// ASSGN  SYS006, DISK, VOL=vvvvvv, SHR
// ASSGN  SYS007, DISK, VOL=vvvvvv, SHR
// DLBL   IJSYS05, 'work5', 1, SD
// EXTENT SYS005, vvvvvv, 1, 0, strk, ntrks
// DLBL   IJSYS06, 'work6', 1, SD
// EXTENT SYS006, vvvvvv, 1, 0, strk, ntrks
// DLBL   IJSYS07, 'work7', 1, SD
// EXTENT SYS007, vvvvvv, 1, 0, strk, ntrks
// DLBL   SYSUT1, 'CEE001'
// EXTENT SYSnnn, vvvvvv
// ASSGN  SYSnnn, DISK, VOL=vvvvvv, SHR
// OPTION CATAL, NODECK
// EXEC  EDCCOMP, SIZE=EDCCOMP, PARM='/INFILE(DD:SYSnnn-SYSUT1), X
        NAME(CEE001)'
/*
* LNKEDT
  INCLUDE urtname
  ENTRY BEGIN
```

```
// EXEC LNKEDT, PARM='AMODE=31, RMODE=ANY'  
/*  
/&  
* $$ E0J
```

## Embedding SQL Statements in Host Programs

You can use SQL statements by embedding them in COBOL, PL/I, Assembler, or C host language programs. Alternatively, some statements can be submitted online through the CA Datacom Datadictionary Interactive SQL Service Facility. See [Statement Execution Table](#). (see page 53)

Detailed information about coding embedded SQL in the various languages can be found as follows:

### **COBOL**

Information begins in [Coding Embedded SQL in COBOL](#) (see page 178).

### **PL/I**

Information begins in [Coding Embedded SQL in PL/I](#) (see page 185).

### **Assembler**

Information begins in [Coding Embedded SQL in Assembler](#) (see page 200).

### **C language**

Information begins in [Coding Embedded SQL in C](#) (see page 206).

When you embed SQL in a host program, your activities include making use of:

1. Preprocessor options (see [Using Preprocessor Options](#) (see page 208)), and
2. SQL statements (see [SQL Statements](#) (see page 597)).

## Distinguishing SQL Statements

To distinguish SQL statements from the host language, you must include special statements in your host program to embed SQL statements.

## COBOL

All embedded SQL statements in COBOL programs are preceded by the keywords EXEC SQL and followed by the keyword END-EXEC as shown in the following format.

```
EXEC SQL
    statement
END-EXEC
```

The alternate format is shown in the following.

```
EXEC SQL statement END-EXEC
```

EXEC SQL and END-EXEC are reserved words for use by the CA Datacom/DB Preprocessor. When the Preprocessor scans the host source, it uses the EXEC SQL and END-EXEC to identify SQL statements to be passed to CA Datacom/DB for preparation.

These keywords must be finished on the same line as they were started. They cannot be continued to another line.

The first character of the EXEC SQL keyword must be the first significant character on that line. The recommended position is column 12 or further right. Do not terminate the EXEC SQL keyword with a period (.). Terminating the EXEC SQL keyword with a period results in a syntax error.

The END-EXEC keyword must be the only or last word on the line. Terminating the END-EXEC keyword with a period (.) is only necessary when a period is required to make the embedded SQL statement consistent with the COBOL logic of your program. The generated COBOL code has a period at the end of the statement *only* if you terminated the END-EXEC with a period. For all other embedded SQL statements (declarative or Preprocessor declarations and instructions) a terminating period after END-EXEC is not necessary and is ignored.

**Note:** See [Executable SQL Statements](#) (see page 356) for information about submitting SQL statements online using the Interactive SQL Service Facility.

## PL/I and C

All embedded SQL statements in PL/I and C programs are preceded by the keywords EXEC SQL and followed by a semicolon:

```
EXEC SQL
    statement
;
```

The identifying words EXEC SQL (which are reserved), the statement, and the terminating semicolon can appear on one or more lines within the prescribed margins. Do not include code unrelated to SQL on these lines since the Preprocessor comments them. The word EXEC must be separated from the word SQL, and the word SQL must be separated from the statement. The semicolon can directly follow the last letter of the statement.

### Example 1

```
EXEC SQL
    DECLARE ADDR_CURSOR CURSOR FOR
    SELECT ID, NAME_LAST, STATE FROM ADDR_TBL;
```

### Example 2

```
EXEC SQL
    WHENEVER NOT FOUND GOTO ALL_DONE ;
```

### Example 3

```
EXEC SQL
    FETCH ADDR_CURSOR
    INTO :HID, :HNAME_LAST, :HSTATE;
```

### Example 4

```
EXEC SQL CLOSE ADDR_CURSOR;
```

## Assembler

All embedded SQL statements in Assembler programs are preceded by the keywords EXEC SQL and ended on a record for which the continuation field is not used.

```
EXEC SQL                                X
    statement                            X
    statement
```

The identifying words EXEC SQL (which are reserved), and the statement can appear on one or more lines within the prescribed margins. Do not include code unrelated to SQL on these lines since the SQL Preprocessor for Assembler comments them. The word EXEC must be separated from the word SQL, and the word SQL must be separated from the statement.

### Example 1

```
EXEC SQL                                X
    DECLARE ADDR_CURSOR CURSOR FOR      X
    SELECT ID, NAME_LAST, STATE FROM ADDR_TBL
```

### Example 2

```
EXEC SQL                                X
    WHENEVER NOT FOUND GOTO ALLDONE
```

### Example 3

```
EXEC SQL                                X
    FETCH ADDR_CURSOR                  X
    INTO :HID, :HNAME_LAST, :HSTATE
```

### Example 4

```
EXEC SQL CLOSE ADDR_CURSOR
```

## Rules for Coding Embedded SQL

When you embed SQL statements in your host program, you must adhere to the following requirements:

- Exactly one SQL statement is allowed:
  - In COBOL, between a pair of EXEC SQL and END-EXEC keywords.
  - In PL/I and C, between the EXEC SQL and the terminating semicolon.
  - In Assembler, for an EXEC SQL.
- In COBOL, BEGIN DECLARE SECTION and END DECLARE SECTION can *only* be used in the WORKING-STORAGE SECTION of your program.
- In COBOL, INCLUDE is *not* recognized by the ANSI standard.



- Host-variable requirements are:
  - The maximum length allowed for host-variables for all languages is 32,765 bytes.
  - In COBOL, all host-variables in your program must be declared before you code any DECLARE cursor-name CURSOR statement.
  - In PL/I and C, a host-variable in your program must be declared in the source prior to its reference by an SQL statement, and you must determine the proper scope for host-variables.
  - In Assembler, you must determine the proper scope for host-variables.
  - for the C language, also see [Rules for Coding Host-Variables in C](#) (see page 206).
- In COBOL, all other embedded SQL statements must be in the PROCEDURE DIVISION.
- To conform to the ANSI standards for embedded SQL, exactly one OPEN and one CLOSE must be coded:
  - In COBOL, for each defined cursor name (each may be executed multiple times by the program). For extended mode SQL, at least one OPEN and one CLOSE must be coded for each defined cursor. This means that you are responsible for correctly coding the logic of OPEN and CLOSE.
  - In PL/I and Assembler, for each cursor defined.
- In PL/I and Assembler, WHENEVER statements must appear in the source prior to their use. Any previous WHENEVER is in effect until replaced.

## Additional Assembler Requirements

When you embed SQL statements in an Assembler program, the program must provide an area of SQLDSIZ and establish addressability by using the DSECT SQLDSECT. Both SQLDSIZ and SQLDSECT are generated by the Preprocessor.

### **SQLDSECT**

is a DSECT of the storage areas which are required for SQL use.

### **SQLDSIZ**

is a fullword where the value is the length required for storage used by SQL. SQLDSIZ is variable in size according to the type and number of SQL statements, the number of host variables, and whether the SQLCA is generated separately. The area must be initialized to binary zeros prior to the first call for SQL.

In Assembler, registers 0, 1, 14, and 15 are used in generated code. Do not rely on their contents being consistent where code has been generated.

## Coding Embedded SQL in COBOL

The COBOL versions supported are:

<b>z/OS</b>	<b>z/VSE</b>
OS/VS COBOL 2.4	DOS/VS COBOL 3.0
VS/COBOL II 3.0	VS/COBOL II 3.0

The following sections describe the code which must be included in each COBOL division. The ENVIRONMENT DIVISION is not listed since there is no applicable SQL code in that division.

### IDENTIFICATION DIVISION

The CA Datacom/DB SQL Preprocessor uses the COBOL PROGRAM-ID as the name for the plan associated with your program, unless another name is specified in the Preprocessor options. If you use the PROGRAM-ID for the plan name, then the name should be unique within your authorization ID.

### DATA DIVISION

Following are the requirements for embedding SQL in the DATA DIVISION of your program.

### WORKING-STORAGE SECTION

The WORKING-STORAGE SECTION statement must be coded on the same line with no continuation.

### Host-Variable Definitions in COBOL

Host-variables can be defined only in the WORKING-STORAGE SECTION of your DATA DIVISION. See Host Variables for rules on naming host-variables. Host-variable definitions:

- Cannot be longer than 32,765 bytes.
- Cannot be COMPUTATIONAL-1 or COMP-1 data type.
- Cannot have a PICTURE clause containing the following editing symbols:

Z	\$ (dollar sign)	. (period)
B	* (asterisk)	+ (plus sign)
0 (zero)	, (comma)	- (minus sign)

If the CA Datacom/DB Preprocessor encounters any host-variable which includes any of the above clauses, data types or symbols, it issues an error message and the variable is not stored in the host-variable symbol table.

The following sections describe ANSI and extended mode requirements for host-variables.

**ANSI SQL Standard Host-Variable Definitions**

If you specify in the CA Datacom/DB SQL Preprocessor options that all SQL statements must be ANSI standard, then all host-variables:

- Must be defined in one or more SQL DECLARE sections
- Must be defined at the elementary column level

All host-variables do not have to be defined within the same SQL DECLARE section. However, any host-variable you define must appear in only one SQL DECLARE section. Host-variables defined in an ANSI SQL DECLARE section become standard COBOL variables after processing by the Preprocessor. The form of an ANSI SQL DECLARE section is:

EXEC SQL

BEGIN DECLARE SECTION

END-EXEC  
 host-variable definitions  
 EXEC SQL

END DECLARE SECTION

END-EXEC

**ANSI Standard Host-Variable Data Types**

The ANSI standard allowable data types are as follows:

SQL Data Type	COBOL Definition
CHARACTER or CHAR	PIC X(n)
NUMERIC (zoned decimal)	PIC S9(n)V9(n) DISPLAY SIGN LEADING SEPARATE

SQL Data Type	COBOL Definition
INTEGER or INT (large integer)	PIC S9(n) COMP where n > 4 and n <= 9 is supported n > 9 is not supported

#### Extended Mode SQL Host-Variable Definitions

Host-variables can appear outside an SQL DECLARE section if you specify in the CA Datacom/DB SQL Preprocessor options that SQL statements can include CA Datacom/DB extensions.

#### Non-ANSI Host-Variable Data Types

The non-ANSI standard allowable data types are as follows:

SQL Data Type	COBOL Definition
CHARACTER or CHAR	PIC X(n)
NUMERIC (zoned decimal)	PIC 9(n)V9(n) DISPLAY or PIC S9(n)V9(n) DISPLAY or PIC S9(n)V9(n) DISPLAY SIGN LEADING SEPARATE
DECIMAL or DEC (packed decimal)	PIC 9(n)V9(n) COMP-3 or PIC S9(n)V9(n) COMP-3 or PIC 9(n)V9(n) PACKED-DECIMAL or PIC S9(n)V9(n) PACKED-DECIMAL
SMALLINT (small integer)	PIC S9(n) COMP or PIC S9(n) BINARY or PIC S9(n) COMP-4 where n <= 4
INTEGER or INT (large integer)	PIC S9(n) COMP or PIC S9(n) BINARY or PIC S9(n) COMP-4 where n > 4 and n <= 9 is supported n > 9 is not supported
FLOAT	USAGE COMP-2
REAL	USAGE COMP-2
DOUBLE PRECISION	USAGE COMP-2

**Note:** Variable-name-length and variable-name-text are user-defined names. The required level for these two VARCHAR elementary items is 49. The group level may be numbered 01 through 48.

SQL Data Type	COBOL Definition
VARCHAR (small integer) (character)	49 variable-name-length PIC S9(4) COMP  49 variable-name-text PIC X(n) where n = maximum length
LONG VARCHAR (small integer) (character)	49 variable-name-length PIC S9(4) COMP  49 variable-name-text PIC X(n) where n = maximum length
DATE	PIC X(10)
TIME	PIC X(8)
TIMESTAMP	PIC X(26)
GRAPHIC	PIC G(n) USAGE DISPLAY-1 or PIC N(n) USAGE DISPLAY-1
VARGRAPHIC (small integer) (character)	49 variable-name-length PIC S9(4) COMP  49 variable-name-text PIC G(n) USAGE DISPLAY-1 or PIC N(n) USAGE DISPLAY-1 where n = maximum length  <b>Note:</b> Variable-name-length and variable-name-text are user-defined names. The required level for these two VARGRAPHIC elementary items is 49. The group level may be numbered 01 through 48. See GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC.
LONG VARGRAPHIC (small integer) (character)	49 variable-name-length PIC S9(4) COMP  49 variable-name-text PIC G(n) USAGE DISPLAY-1 or PIC N(n) USAGE DISPLAY-1 where n = maximum length  <b>Note:</b> Variable-name-length and variable-name-text are user-defined names. The required level for these two LONG VARGRAPHIC elementary items is 49. The group level may be numbered 01 through 48. See GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC.

**Note:** The lengths shown for DATES, TIMES, and TIMESTAMPS are minimum length requirements. See [Character String Literals](#) (see page 510) for more information.

The CA Datacom/DB Preprocessor for COBOL recognizes GRAPHIC host variables and allows mixed data in literals. The Shift-Out and Shift-In characters specified in the CXXMAINT option of the CA Datacom/DB Utility (DBUTLTY) are used by the Preprocessor. Support is provided for both the IBM COBOL and COBOL II compilers and the Fujitsu COBOL85 compiler, but only in COBOL can host-variable names include DBCS characters (delimited by Shift characters).

For more information about SQL data types, see [Data Types](#) (see page 485).

## PROCEDURE DIVISION

The PROCEDURE DIVISION statement must be coded on the same line with no continuation.

The following sections discuss the requirements for embedding SQL in the PROCEDURE DIVISION of your program.

### Declaring Cursors in COBOL

Cursors must be declared in the source:

- Before any reference to the cursor, and
- After all host-variables used in the definition have been defined.

Cursor declarations can be made in the:

- WORKING-STORAGE SECTION
- LINKAGE SECTION
- REPORT SECTION
- PROCEDURE DIVISION

The recommended practice is to place all cursor definitions immediately before the PROCEDURE DIVISION statement.

Cursor definitions are declarations, not action (procedural) statements.

### Declaring Exceptions in COBOL

An exception declaration specifies a COBOL action to be taken when an exception occurs during execution of an SQL statement. The form of the exception declaration is:

```
EXEC SQL
    WHENEVER condition exception-action
END-EXEC
```

Following are the exception conditions for ANSI mode and extended mode.

<b>ANSI</b>	<b>Extended Mode</b>
SQLERROR	SQLERROR
NOT FOUND	NOT FOUND
	SQLWARNING

Following are the exception actions for ANSI mode and extended mode.

<b>ANSI</b>	<b>Extended Mode</b>
GOTO target	GOTO target
GO TO target	GO TO target
CONTINUE	CONTINUE

GOTO and GO TO cause transfer of control to the target paragraph or section.

CONTINUE causes the program to continue execution with the next COBOL statement.

The scope of an exception declaration:

- Begins with the placement of the exception declaration in the source, and
- Continues until either another exception declaration for the same condition in the source or the end of the source occurs.

**Note:** SQL statements occurring in the source before an exception declaration are not affected by it.

Any SQL statements executed that are not under control of an exception declaration default to CONTINUE.

If an exception declaration is not provided, the recommended practice is that your program include code to check the SQLCODE value in the SQL Communication Area (SQLCA) immediately after each executable SQL statement.

### Executable SQL Statements in COBOL

Executable SQL statements must appear in the PROCEDURE DIVISION in the following format:

```
EXEC SQL
```

```
    statement
```

```
END-EXEC
```

Executable embedded SQL statements can appear anywhere that a COBOL statement can appear.

### INCLUDE Directive in COBOL

You can use the SQL INCLUDE directive to get secondary input from the include library, INCLUDE. The INCLUDE directive causes input to be read from the specified member of the include library until the end of the member is reached. The included library input cannot contain other INCLUDE directives, but can contain both host language and SQL statements. The include library must have fixed-length records of 80 bytes.

You can use the INCLUDE only in the extended mode. If the Preprocessor finds an INCLUDE while in the ANSI mode, it ignores the INCLUDE during processing, and issues a warning message. Processing by the Preprocessor continues.

The INCLUDE can be used anywhere within the program. The Preprocessor locates the member name and includes the member in the COBOL source. If the result is invalid COBOL, the precompiler does not check for valid COBOL syntax in an INCLUDE.

**Note:** INCLUDE directives cannot be nested, that is to say, a member named in an INCLUDE directive cannot contain an INCLUDE directive.



**z/OS**

In z/OS, the format for the INCLUDE instruction is:

```
EXEC SQL
    INCLUDE member-name
END-EXEC
```

**z/VSE**

In z/VSE, the format for the INCLUDE instruction is:

```
EXEC SQL
    INCLUDE member-name,member-type
END-EXEC
```

**Note:** In the previous example, member-type is a one-byte identifier of the file type under which the member is cataloged. Any letter is a valid value for member-type. If member-type is not specified, it defaults to C.

## Coding Embedded SQL in PL/I

### Distinguishing Storage for Use as Host-Variables in PL/I

The information in this section is divided into three parts:

- ANSI and DB2A86 modes
- Modes other than ANSI and DB2A86
- All modes

**For ANSI and DB2A86 Modes**

For ANSI and DB2A86 SQL modes, all storage that is referenced by the statement must be delimited by starting and ending statements. For example:

```
EXEC SQL BEGIN DECLARE SECTION ;
    storage declarations of host-variables.
EXEC SQL END DECLARE SECTION ;
```

In the previous example, the EXEC SQL lines (through and including the semicolon) are reserved. They may appear on one or more lines but should not be mixed with code not related to SQL since the Preprocessor comments the lines. You can, if desired, place the semicolon immediately following SECTION (that is to say, without a space between the N and the semicolon). All other reserved words must be separated by a space.

You may have more than one SQL declare section. All references to storage must follow the declaration in the source.

You must comply with the following requirements when you code host-variable declarations:

- All variables must be declared at the elementary level. You may declare more than one occurrence of the same variable type in a declare.
- All declarations within the DECLARE SECTION must conform as either a potential host or indicator.

#### **For Modes Other Than ANSI and DB2A86**

For modes other than ANSI and DB2A86, you must comply with the following requirements when you code host-variable declarations:

- You may declare structures of fields. These may be referenced as direct or indirect.

##### **direct**

A direct reference specifies a field within a structure or substructure that can be used as a host-variable. A reference in this form may be the name or a qualified name of two levels.

##### **indirect**

Indirect reference specifies a structure or substructure that can be expanded in its entirety into fields that are host-variables. A reference to a structure is its name. A reference to a substructure is its name which must be unique or a unique two-level qualified name.

For example, consider the following abstract structures. Assume elementary fields except X are eligible to be host fields.

Structure 1

```
DCL 1  A,  
      2  B  ... ,  
      2  C  ... ;
```

Structure 2

```
DCL 1  D,  
      2  E,  
      3  F  ... ,  
      3  G  ... ,  
      2  H  ... ;
```

Structure 3

```
DCL 1  M,  
      2  N  ... ,  
      2  O  ... ,  
      2  X  ... ;
```

Structure 1 can be referenced directly by specifying B, C, A.B or A.C. It can be referenced indirectly by specifying A which is expanded into its components B and C to match the SQL statement.

Structure 2 can be referenced directly by specifying F, G, or H. Structure D cannot be referenced indirectly since it is more than two levels. Substructure E can be referenced indirectly expanding into fields F and G.

Structure 3 can be referenced directly by specifying N or O. Remember that X is invalid as a host-variable. Although structure M is two levels, indirect reference is not possible since X is a component.

An array of indicator variables may be specified with an indirect reference. Each element field is paired with an indicator variable until all fields have indicators or there are no more indicators remaining in the array.

- Declarations used as host-variables or indicators may appear after a statement in the source. The use of EXEC SQL BEGIN and END DECLARE SECTION markers is optional.

**For All Modes**

For all modes, you must comply with the following requirements when you code host-variable declarations:

- Indicator variables must be declared as fixed binary(15) or binary fixed(15). As with host-variables, declarations of indicator variables may contain multiple occurrences.
- Declarations may be defined in PL/I INCLUDE directives that have been expanded by the Preprocessor.
- Declarations may be defined in SQL INCLUDE directives.

Host-Variable Declarations for PL/I

The maximum length allowed for a host-variable is 32,765 bytes.

**ANSI Standard Host-Variable Data Types in PL/I**

The ANSI standard allowable data types are as follows:

SQL Data Type	PL/I Declaration
CHARACTER or CHAR	CHARACTER(n) or CHAR(n)
DECIMAL or DEC (packed decimal)	FIXED DECIMAL (p,s) or DECIMAL FIXED (p,s) or FIXED (p,s) DECIMAL or DECIMAL (p,s) FIXED
INTEGER or INT (large integer)	FIXED BINARY (n) or BINARY FIXED (n) or FIXED (n) BINARY or BINARY (n) FIXED where 16 <= n <= 31

SQL Data Type	PL/I Declaration
FLOAT	BINARY FLOAT (n) or FLOAT BINARY (n) or BIN FLOAT (n) or FLOAT BIN (n)      where 22 <= n <= 53

Valid short forms of the declarations are:

- FIX for FIXED
- DEC for DECIMAL
- BIN for BINARY
- CHAR for CHARACTER

#### Non-ANSI Host-Variable Data Types in PL/I

The non-ANSI standard allowable data types are as follows:

SQL Data Type	PL/I Declaration
CHARACTER or CHAR	CHARACTER(n) or CHAR(n)
NUMERIC (zoned decimal)	PICTURE(pVs) or PIC(pVs)
DECIMAL or DEC (packed decimal)	FIXED DECIMAL (p,s) or DECIMAL FIXED (p,s) or FIXED (p,s) DECIMAL or DECIMAL (p,s) FIXED
SMALLINT (small integer)	FIXED BINARY (n) or BINARY FIXED (n) or FIXED (n) BINARY or BINARY (n) FIXED    where 1 <= n <= 15
INTEGER or INT (large integer)	FIXED BINARY (n) or BINARY FIXED (n) or FIXED (n) BINARY or BINARY (n) FIXED    where 16 <= n <= 31
FLOAT	BINARY FLOAT (n) or FLOAT BINARY (n) or BIN FLOAT (n) or FLOAT BIN (n)      where 22 <= n <= 53 DECIMAL FLOAT (n) or DEC FLOAT (n)    where 7 <= n <= 16
DATE	CHARACTER(10) or CHAR(10)
TIME	CHARACTER(8) or CHAR(8)
TIMESTAMP	CHARACTER(26) or CHAR(26)
VARCHAR	CHARACTER(n) VARYING
LONG VARCHAR	CHARACTER(n) VARYING
GRAPHIC	GRAPHIC(x) where x is the precision (maximum number of DBCS characters)
VARGRAPHIC	GRAPHIC(x) VARYING where x is the precision (maximum number of DBCS characters)

SQL Data Type	PL/I Declaration
LONG VARGRAPHIC	GRAPHIC(x) VARYING where x is the precision (maximum number of DBCS characters)

Valid short forms of the declarations are:

- FIX for FIXED
- DEC for DECIMAL
- BIN for BINARY
- CHAR for CHARACTER
- **Exception: With VARCHAR, do not use the short form of CHARACTER.**

The CA Datacom/DB Preprocessor for PL/I recognizes GRAPHIC host variables and allows mixed data in literals. The Shift-Out and Shift-In characters specified in the CXXMAINT option of the CA Datacom/DB Utility (DBUTLTY) are used by the Preprocessor.

**Note:** PL/I controlled variables are not supported.

### PL/I Examples

In the following examples, BEGIN and END DECLARE SECTION markers are used for clarity. They may or may not be required, depending on the SQL mode being used (see [Distinguishing Storage for Use as Host-Variables in PL/I](#) (see page 185)).

#### PL/I Example 1

```
EXEC SQL BEGIN DECLARE SECTION ;
          /*-----
            Host-variables
          *-----*/
DECLARE HID          CHAR(12)          INIT(' ');
DECLARE HNAME_LAST  CHAR(30)          INIT(' ');
DECLARE HSTATE      CHAR(36)          INIT(' ');
          /*-----
            Indicator variables
          *-----*/
DECLARE IID          FIXED BINARY(15)  INIT(0);
DECLARE INAME_LAST  FIXED BINARY(15)  INIT(0);
DECLARE ISTATE      FIXED BINARY(15)  INIT(0);
EXEC SQL END DECLARE SECTION ;
```

*PL/I Example 2*

```

/*-----
   An alternate form for the declare
   section.
   *-----*/
EXEC SQL
  BEGIN DECLARE SECTION ;
%SKIP
/*-----
   Indicator & host-variable pair
   *-----*/

DCL ID_IVAR      FIXED BINARY(15);
DCL ID           CHAR(12);
%SKIP
DCL NAME_LAST_IVAR FIXED BINARY(15);
DCL NAME_CODE    CHAR(30);
%SKIP
EXEC SQL
  END DECLARE SECTION ;

```

*PL/I Example 3*

```

EXEC SQL BEGIN DECLARE SECTION ;
/*-----
   Host-variables
   *-----*/
DECLARE HID          CHAR(12)      INIT(' ');
DECLARE HNAME_LAST   CHAR(30)      INIT(' ');
DECLARE HSTATE       CHAR(36)      INIT(' ');
%SKIP
/*-----
   Multiple indicator variables in
   a single declare
   *-----*/
DECLARE ( IID,
         INAME_LAST,
         ISTATE )    FIXED BINARY(15) INIT(0);
EXEC SQL END DECLARE SECTION ;

```

*PL/I Example 4*

```
EXEC SQL BEGIN DECLARE SECTION ;
```

```
/*-----  
Host-variables within a structure
```

Possible direct references:

Name	Qualified name
-----	-----
SEQ	GRPED.SEQ
NAME_FIRST	NAME_ALL.NAME_FIRST
NAME_MIDDLE	NAME_ALL.NAME_MIDDLE
NAME_LAST	NAME_ALL.NAME_LAST
NAME_ADDL	NAME_ALL.NAME_ADDL
SSAN	GRPED.SSAN
CITY	GRPED.CITY
STATE	GRPED.STATE
ZIP_5	ZIP.ZIP_5
ZIP_4	ZIP.ZIP_4

Possible indirect references:

NAME\_ALL

Ineligible fields and references:

Specification	Reason
ZIP_ST	not 15,0 or 31,0 for use as short integer or integer respectively
NOTES	array
GRPED	too many levels & contains ineligible field
ZIP	ineligible field

\*-----\*/

DECLARE

```

1  GRPED,
2  SEQ          CHAR(2),
2  NAME_ALL ,
3  NAME_FIRST  CHAR(30),
3  NAME_MIDDLE CHAR(30),
3  NAME_LAST   CHAR(30),
3  NAME_ADDL   CHAR(30),
2  SSAN        CHAR(11),
2  CITY        CHAR(30),
2  STATE       CHAR(30),
2  ZIP,
3  ZIP_5       PIC '9999T' ,
3  ZIP_4       PIC '999T' ,
3  ZIP_ST      FIXED BIN(9,4),
2  NOTES(3)    CHAR(80);
    
```

```

DECLARE (ISEQ, INAME, ISSAN, ICITY, ISTATE, IZIP )
        FIXED BINARY(15) INIT(0);
    
```

EXEC SQL END DECLARE SECTION ;



*PL/I Example 5*

```

EXEC SQL BEGIN DECLARE SECTION ;
/*-----
Host-variables within a structure

Possible indirect references:
NAME
ZIP

Invalid indirect references:
Specification Reason
-----
ADDRESS      too many levels
              & contains array
TYPE          contains array
MAILING       too many levels

When CHAR(n) VARYING is specified,
the two fields generated do not
count as an additional level for
expansion purposes.
*-----*/

DECLARE
1 ADDRESS,
2 TYPE,
3 TYPE_R      CHAR(1),
3 TYPE_X(3)   CHAR(1),
2 NAME,
3 FIRST_NAME  CHAR(30) VARYING,
3 MIDDLE_NAME CHAR(30) VARYING,
3 LAST_NAME   CHAR(30) VARYING,
2 MAILING,
3 CARE_OF     CHAR(60) VARYING,
3 LINE_1      CHAR(60) VARYING,
3 LINE_2      CHAR(60) VARYING,
3 LINE_3      CHAR(60) VARYING,
3 CITY        CHAR(60),
3 STATE       CHAR(30),
3 ZIP,
4 ZIP_5       PIC'9999T',
4 ZIP_4       PIC'999T';

EXEC SQL END DECLARE SECTION ;

```

*PL/I Example 6*

```

EXEC SQL BEGIN DECLARE SECTION ;
                                     /*-----
                                     Host-variables within a structure

                                     This structure, though similar to
                                     the previous example, expands
                                     entirely.
                                     *-----*/

DECLARE
  1  ADDRESS,
  2  TYPE           CHAR(4),
  2  FIRST_NAME    CHAR(30) VARYING,
  2  MIDDLE_NAME   CHAR(30) VARYING,
  2  LAST_NAME     CHAR(30) VARYING,
  2  CARE_OF       CHAR(60) VARYING,
  2  LINE_1        CHAR(60) VARYING,
  2  LINE_2        CHAR(60) VARYING,
  2  LINE_3        CHAR(60) VARYING,
  2  CITY          CHAR(60),
  2  STATE         CHAR(30),
  2  ZIP           PIC'99999999T';

EXEC SQL END DECLARE SECTION ;

```

## Rules for SQL INCLUDEs in PL/I

The format for an SQL INCLUDE directive is similar to other SQL statements:

```
EXEC SQL INCLUDE member-name ;
```

**Note:** INCLUDEs in the C language are basically the same as in PL/I, but see the exceptions for C noted in [INCLUDEs in C](#) (see page 150).

Each SQL INCLUDE may specify only one member name contained in the INCLUDE data set. You may, however, code more than one SQL INCLUDE.

INCLUDEs cannot include other INCLUDEs. If an SQL INCLUDE is embedded in an SQL INCLUDE, the Preprocessor notes the error and does not read its contents. If a PL/I INCLUDE is embedded in the SQL INCLUDE, the Preprocessor ignores it.

Except for SQLDA, member names that begin with the letters SQL are not included because these are assumed to be control block names. The EXEC SQL INCLUDE statement is, however, commented. When INCLUDE SQLDA is specified, the Preprocessor for PL/I includes the description of a SQL Descriptor Area (SQLDA) for use by dynamic SQL statements.

SQL INCLUDEs are processed in a special manner by the Preprocessor. If the Preprocessor is able to open the INCLUDE, the records contained in the INCLUDE are inserted in the source after the semicolon of the SQL INCLUDE. These records are processed as if they were source except in the case of an INCLUDE within an INCLUDE, as mentioned above.

In the modified source the SQL INCLUDE directive is commented.

The following examples illustrate the inclusion of host declarations from INCLUDEs. In one case the SQL declaration section statements are in the INCLUDE itself. The other INCLUDE contains only PL/I declarations.

For z/OS, the name in the EXEC SQL INCLUDE is the name of the member in the partition data set of the INCLUDE DD.

For z/VSE, the name in the EXEC SQL INCLUDE is part of the member name in the z/VSE library. The other part, the file or book type, is the letter P or any letter designated by the ITYP= Preprocessor option. The DLBL name must be INCLUDE.

EXEC SQL INCLUDE SQLCA causes the SQLCA to be generated at that point in the source rather than with other SQL request blocks.

#### PL/I Example 1

Following is the source:

```
/*-----  
    This INCLUDE contains the statements for  
    an SQL declare section.  
*-----*/  
EXEC SQL INCLUDE PL1INCT1 ;
```

Following is the report for the source after the INCLUDE is read. The INCLUDE source is denoted by a plus sign after the sequence number. The sequence number is the record number for the respective file (source or INCLUDE).

**Report for Source after INCLUDE is Read (Example 1— PL/I)**

```

34
35          /*-----
36          This INCLUDE contains the statements for
37          an SQL declare section.
38          *-----*/
39 EXEC SQL INCLUDE PL1INCT1 ;
1+ /*-----
2+ test include with storage only
3+ *-----*/
4+
5+ EXEC SQL BEGIN DECLARE SECTION ;
6+   DECLARE CHSEQ CHAR(2)   INIT('02'); /* host */
7+   DECLARE HSEQ  CHAR(2)   INIT(' '); /* host */
8+   DECLARE HNAME CHAR(20)  INIT(' '); /* host */
9+   DECLARE HSSAN CHAR(11)  INIT(' '); /* host */
10+  DECLARE HCITY  CHAR(10)  INIT(' '); /* host */
11+  DECLARE HZIP   PIC'9999T'; /* host */
12+  DECLARE (ISEQ, INAME, ISSAN, ICITY, IZIP )
13+          FIXED BINARY(15) INIT(0); /* indicator */
14+ EXEC SQL END DECLARE SECTION ;
15+ /* end include */
40

```

Following is the report for the modified source that is used as input to the compiler:

### Report for the Modified Source (Example 1— PL/I)

```

34
35          /*-----
36          This INCLUDE contains the statements for
37          an SQL declare section.
38          *-----*/
/* commented by CA-DATACOM/DB Preprocessor */
39 EXEC SQL INCLUDE PL1INCT1 ;
/* commented by CA-DATACOM/DB Preprocessor */
1+ /*-----
2+ test include with storage only
3+ *-----*/
4+
/* commented by CA-DATACOM/DB Preprocessor */
5+ EXEC SQL BEGIN DECLARE SECTION ;
/* commented by CA-DATACOM/DB Preprocessor */
6+ DECLARE CHSEQ CHAR(2) INIT('02'); /* host */
7+ DECLARE HSEQ CHAR(2) INIT(' '); /* host */
8+ DECLARE HNAME CHAR(20) INIT(' '); /* host */
9+ DECLARE HSSAN CHAR(11) INIT(' '); /* host */
10+ DECLARE HCITY CHAR(10) INIT(' '); /* host */
11+ DECLARE HZIP PIC'9999T'; /* host */
12+ DECLARE (ISEQ, INAME, ISSAN, ICITY, IZIP )
13+         FIXED BINARY(15) INIT(0); /* indicator */
/* commented by CA-DATACOM/DB Preprocessor */
14+ EXEC SQL END DECLARE SECTION ;
/* commented by CA-DATACOM/DB Preprocessor */
15+ /* end include */
40

```

### PL/I Example 2

Following is the source:

```

/*-----
This INCLUDE contains only declarations.
*-----*/
EXEC SQL BEGIN DECLARE SECTION ;
EXEC SQL INCLUDE PL1INCT2 ;
EXEC SQL END DECLARE SECTION ;

```

Following is the report for the source after the INCLUDE is read:

**Report for Source after INCLUDE is Read (Example 2— PL/I)**

```
41          /*-----
42          This include contains only declarations.
43          *-----*/
44 EXEC SQL BEGIN DECLARE SECTION ;
45 EXEC SQL INCLUDE PL1INCT2 ;
1+ /*-----
2+ test include with storage only within declare section
3+ *-----*/
4+
5+ DECLARE CHSEQ CHAR(2) INIT('02'); /* host */
6+ DECLARE HSEQ CHAR(2) INIT(' '); /* host */
7+ DECLARE HNAME CHAR(20) INIT(' '); /* host */
8+ DECLARE HSSAN CHAR(11) INIT(' '); /* host */
9+ DECLARE HCITY CHAR(10) INIT(' '); /* host */
10+ DECLARE HZIP PIC'9999T'; /* host */
11+ DECLARE (ISEQ, INAME, ISSAN, ICITY, IZIP )
12+         FIXED BINARY(15) INIT(0); /* indicator */
13+ /* end include */
46 EXEC SQL END DECLARE SECTION ;
47
```

Following is the report for the modified source that is used as input to the compiler:

**Report for the Modified Source (Example 2— PL/I)**

```

41          /*-----
42          This INCLUDE contains only declarations.
43          *-----*/
/* commented by CA-DATACOM/DB Preprocessor */
44 EXEC SQL BEGIN DECLARE SECTION ;
45 EXEC SQL INCLUDE PL1INCT2 ;
/* commented by CA-DATACOM/DB Preprocessor */
1+ /*-----
2+ test include with storage only within declare section
3+ *-----*/
4+
5+ DECLARE CHSEQ CHAR(2) INIT('02'); /* host */
6+ DECLARE HSEQ CHAR(2) INIT(' '); /* host */
7+ DECLARE HNAME CHAR(20) INIT(' '); /* host */
8+ DECLARE HSSAN CHAR(11) INIT(' '); /* host */
9+ DECLARE HCITY CHAR(10) INIT(' '); /* host */
10+ DECLARE HZIP PIC'9999T'; /* host */
11+ DECLARE (ISEQ, INAME, ISSAN, ICITY, IZIP )
12+ FIXED BINARY(15) INIT(0); /* indicator */
13+ /* end include */
/* commented by CA-DATACOM/DB Preprocessor */
46 EXEC SQL END DECLARE SECTION ;
/* commented by CA-DATACOM/DB Preprocessor */
47

```

**PL/I Example 3**

Following is an example of the source:

```

/*-----
This INCLUDE contains an SQL open statement.
*-----*/
EXEC SQL INCLUDE PL1INCT6 ;

```

Following is the report for the source after the INCLUDE is read:

**Report for Source after INCLUDE is Read (Example 3— PL/I)**

```

87          /*-----
88          This include contains an SQL open statement.
89          *-----*/
90 EXEC SQL INCLUDE PL1INCT6 ;
1+ EXEC SQL OPEN C1
2+ ;
91

```

Following is the report for the modified source that is used as input to the compiler:

**Report for the Modified Source (Example 3— PL/I)**

```
87          /*-----  
88          This INCLUDE contains an SQL open statement.  
89          *-----*/  
/* commented by CA-DATACOM/DB Preprocessor */  
90 EXEC SQL INCLUDE PL1INCT6 ;  
1+ EXEC SQL OPEN C1  
2+ ;  
   * commented by CA-DATACOM/DB Preprocessor */  
/* start of CA-DATACOM/DB Preprocessor generation */  
   IF XYZK6_DBPRIME = XYZK6_DBPRIME_YES  
   THEN CALL XYZK6_DBINIT;  
   CALL DBSQL( SQLCA, SQLWA0);  
/* end of CA-DATACOM/DB Preprocessor generation */  
91
```

## Coding Embedded SQL in Assembler

### Rules for Coding Host Variables in Assembler

When you code host variable declarations, you must comply with the following requirements:

- Cannot be longer than 32,765 bytes.
- Indicator variables must be declared as a halfword with implied or explicit length of two.
- You may declare structures of fields. Only elementary level fields can be referenced as host variables.
- Definitions may be defined in SQL INCLUDE directives.
- Host and indicator fields must have a duplication factor of one, explicit or implied.
- Fixed format is not required.
- All eligible field definition statements must have a name entry. Use of the prefix SQL should be avoided.
- The operation entry can be either a DS or DC.
- The operand entry must define a single field with the exception noted below.



### Operand Subfields

The operand subfields:

1. The duplication factor must be 1, regardless of whether it is explicitly specified.
2. Valid types are listed below with the corresponding SQL type.
3. Modifier use is limited primarily to length. Where the explicit length is specified and a nominal value is given, the explicit length takes precedence.
4. Nominal values can be used to calculate length implicitly where no length modifier is given. A nominal value may be scanned to determine the number of decimal places.

### Host Variable Declarations for Assembler

In the table below a lowercase n is a number. A lowercase v is a nominal value.

SQL Data Type	Operand Formats	Notes
CHARACTER	C CLn C'v' CLn'v'	
NUMERIC	Z ZLn Z'v' ZLn'v'	The nominal value is used to determine scale. If no nominal value is given, then the scale is set to zero. For implicit length, the nominal value is used to determine precision and scale. Where both the length modifier and the nominal value are coded, the explicit length from the length modifier is used.
DECIMAL	PLn P'v' PLn'v'	The nominal value is used to determine scale. If no nominal value is given, then the scale is set to zero. For implicit length, the nominal value is used to determine precision and scale. Where both the length modifier and the nominal value are coded, the explicit length from the length modifier is used.
SMALLINT	H HL2 HL2'v'	When the length modifier is coded, it must be two. Any nominal value is ignored.
INTEGER	F FL4 F'v' FL4'v'	When the length modifier is coded, it must be four. Any nominal value is ignored.
FLOAT	D DL8 D'v' DL8'v'	When the length modifier is coded, it must be eight. Any nominal value is ignored.

SQL Data Type	Operand Formats	Notes
DATE	CLn C'vvv.' CLn'v'	The length must be 10 or more.
TIME	CLn C'v' CLn'v'	The length must be eight or more.
TIMESTAMP	CLn C'v' CLn'v'	The length must be 26 or more.
VARCHAR	H,CLn HL2,CLn H'vv',C'v' HL2'v',CLn'v'	The operation must have two and only two operands with the halfword followed by the character. Either, both or neither type may have a nominal value. Likewise a length modifier.
LONG VARCHAR	H,CLn HL2,CLn H'vv',C'v' HL2'v',CLn'v'	The operation must have two and only two operands with the halfword followed by the character. Either, both or neither type may have a nominal value. Likewise a length modifier.
GRAPHIC	DC or DS GLn or G'xxxx' or Gn'xxxx'	
VARGRAPHIC	DC or DS H,GLn	
LONG VARGRAPHIC	DC or DS H,GLn	

**Note:** The CA Datacom/DB Preprocessor for Assembler recognizes GRAPHIC host variables and allows mixed data in literals. The Shift-Out and Shift-In characters specified in the CXXMAINT option of the CA Datacom/DB Utility (DBUTLTY) are used by the Preprocessor.

**Example 1**

```

*-----
* Valid host variables
*-----

LASTNAME DS CL30          . CHAR(30)
POBOX    DS ZL9           . NUMERIC(9) with scale 0
ZIPCODE  DS PL5           . DEC(9,0)
ZIPSPLIT DS P'+99999.9999' . DEC(9,4)

CUR$PAY  DC H'0'         . SMALLINT
CUM$PAID DC F'0'         . INTEGER

INTRATE  DS D            . FLOAT(n)

LACTDATE DC C'          ' . date area
LACTTIME DS CL8          . time area
LTXNSTMP DC CL26' '     . timestamp are

STREET   DS H'0',CL40    . VARCHAR(40)
CITY     DS H,CL30       . VARCHAR(30)
STATE    DS H'0',CL20' ' . VARCHAR(20)

*-----
* Indicator variables
*-----

NAMEFND DC H'0'
ZIPIVAL DS H

```

**Rules for SQL INCLUDEs in Assembler**

The format for an SQL INCLUDE directive is similar to other SQL statements:

```
EXEC SQL INCLUDE member-name
```

Each SQL INCLUDE may specify only one member name contained in the INCLUDE data set. You may, however, code more than one SQL INCLUDE.

INCLUDEs cannot include other INCLUDEs. If an SQL INCLUDE is embedded in an SQL INCLUDE, the Preprocessor for Assembler notes the error and does not read its contents. Likewise, macros are not expanded nor are any other copy types processed.

With the exception of SQLCA and SQLDA, member names that begin with the letters SQL are not included because these are assumed to be control block names. EXEC SQL INCLUDE SQLCA causes separate generation of the SQLCA DSECT. When INCLUDE SQLDA is specified, the Preprocessor for Assembler includes the description of a SQL Descriptor Area (SQLDA) for use by dynamic SQL statements.

SQL INCLUDEs are processed in a special manner by the Preprocessor for Assembler. If the Preprocessor for Assembler is able to open the INCLUDE, the records contained in the INCLUDE are inserted in the source after the SQL INCLUDE. These records are processed as if they were source.

In the modified source the SQL INCLUDE directive is commented.

The following examples illustrate the inclusion of host definitions from INCLUDEs. In one case the SQL declaration section statements are in the INCLUDE itself.

For z/OS, the name in the EXEC SQL INCLUDE is the name of the member in the partition data set of the INCLUDE DD.

For z/VSE, the name in the EXEC SQL INCLUDE is part of the member name in the z/VSE library. The other part, the file or book type, is designated by the ITYP= Preprocessor for Assembler option. The DLBL name must be INCLUDE.

### Assembler Example 1

Following is an example of the source:

```
EXEC SQL INCLUDE ASMINCT3
```

Following is the report for the source after the INCLUDE is read:

```
52      EXEC SQL INCLUDE ASMINCT3
1+*-----
2+*  example of including an SQL statement
3+*-----
4+*
5+  EXEC SQL DECLARE                                X
6+          C1 CURSOR FOR                            X
7+          SELECT SEQ,                              X
8+          NAME                                     X
9+          FROM GACTBLV
10+*
11+*----- end include -----*
53 *
```

Following is the report for the modified source:

**Report for the Modified Source (Example 1— Assembler)**

```

52 *      EXEC SQL INCLUDE ASMINCT3
1+*-----
2+* example of including an SQL statement
3+*-----
4+*
5+* EXEC SQL DECLARE
6+*          C1 CURSOR FOR
7+*          SELECT SEQ,
8+*             NAME
9+*             FROM GACTBLV
10+*
11+*----- end include -----*
53 *

```

**Assembler Example 2**

Following is an example of the source:

```
EXEC SQL INCLUDE ASMINCT4
```

Following is the report for the source after the INCLUDE is read:

**Report for Source after INCLUDE is Read (Example 2— Assembler)**

```

53 *
54      EXEC SQL INCLUDE ASMINCT4
1+ EXEC SQL OPEN C1
55 *

```

Following is the report for the modified source:

**Report for the Modified Source (Example 2— Assembler)**

```

53 *
54 *      EXEC SQL INCLUDE ASMINCT4
1+* EXEC SQL OPEN C1
*----- start of CA-DATACOM/DB generation -----
BAL  14,SQLINIT
LA   1,SQL0
ST   1,SQLPWA
LA   1,SQLPARMS
L    15,SQLVSQLE
BALR 14,15
TM   SQLCODE,X'80'
BO   DUMPNOW
*----- end of CA-DATACOM/DB generation -----
55 *

```

## Coding Embedded SQL in C

### Rules for Coding Host-Variables in C

When you code host-variable declarations in C, you must comply with the following requirements:

- Host and indicator variables must be within the scope of the referencing SQL statement.
- In ANSI mode, host and indicator variables must be within the BEGIN and END DECLARE SECTION. Multiple DECLARE sections are allowed. That is, in C only host variables that are declared within an EXEC SQL BEGIN DECLARE SECTION/EXEC SQL END DECLARE SECTION are valid.

**Note:** The SQLWAs and SQLCA are automatically generated after the first EXEC SQL BEGIN DECLARE SECTION in C.

- Only one level of qualification is allowed, that is, a host or indicator variable can be part of a structure, but that structure may not be nested within another structure.
- Host and indicator variables must be one of the C data types shown in the following table.

**Note:** C does not support VARCHAR as in other languages where the length is specified in a 2-byte binary field in front of the actual value. Instead, C supports VARCHAR as a null-terminated string. This is true for input and output host variables in C. Null-terminated strings can also be used with CHAR columns in C.

#### C Data Types Compared to SQL Data Types

C Data Type	Example	SQL Data Type
single char	char x;	CHAR
char array	char x[10]	VARCHAR
char structure containing exactly one <i>short</i> variable followed by exactly one <i>char</i> array.	char struct { short len1; char text[9]; } hv1;	VARCHAR
decimal(precision, scale)	decimal(5,0)	DECIMAL
short or short int	short hv1;	SMALLINT
int or long int	int hv1;	INTEGER
float	float hv1;	FLOAT
double	double hv1;	FLOAT
char array	char dateHv[11];	DATE

C Data Type	Example	SQL Data Type
char array	char timeHv[9];	TIME
char array	char timeStampHv[23];	TIMESTAMP

### Comments on C Data Types

#### char array

The C char array is treated as a null-terminated string.

char hv1[10] is equivalent to VARCHAR(9).

The VARCHAR length is set based on the number of bytes preceding the null-terminating byte.

Char arrays for DATE, TIME, and TIMESTAMP also need the null-terminating byte.

#### char structure

The char array for the text is *not* null terminated.

#### decimal

Use of decimal requires #include decimal.h (provided by the IBM C Compiler), and the compiler option must *not* be ANSI. Precision and scale is optional. To print decimal values, use (with printf):

```
%D(precision, scale)
```

#### host structures

In DATACOM mode, a host variable structure may be used. The host structure must be a C structure containing only valid host variables. The host structure must not be nested within another structure.

#### indicator variables

In DATACOM mode, a host indicator structure or array may be used. The structure or array must only contain *short* C variables. The indicator structure must not be nested within another structure.

### INCLUDEs

INCLUDEs in the C language are basically the same as in PL/I except note the following:

- The SQLCA is always automatically generated, so no INCLUDE should be used for it. However, you must code your own SQLDA and SQLVAR when using dynamic SQL.
- The EXEC SQL INCLUDE <member name>; SQL statement can be used to include members from the PDS data set INCLUDE in z/OS (for z/VSE, the name in the EXEC SQL INCLUDE is the name of a member in a z/VSE library). INCLUDEs in C cannot be nested.
- Because the Precompiler executes before the C Preprocessor, statements such as #include, #define, and typedef are not expanded. Unless the user executes the language preprocessor only before the CA Datacom/SQL C Preprocessor, host variables in #include files cannot be referenced, and #define and typedefs cannot be used to declare a host variable data type.

## Using Preprocessor Options

In this section:

- For an alphabetically arranged list of the options you can specify, see [Options You Can Specify](#) (see page 214).
- For information specific to COBOL, see [Specifying Processing Options in COBOL](#) (see page 210).
- For information specific to PL/I, C, and Assembler, see [Specifying Processing Options in PL/I, C, and Assembler](#) (see page 211).



## Overview

The Preprocessor is controlled by options which you can specify before submitting a program with embedded SQL.

The values you assign to the options determine how the Preprocessor processes the SQL statements and control certain aspects of the application's environment.

The options you specify build the CA Datacom/DB access plan for your program containing embedded SQL. Every program with embedded SQL must have a plan, which is unique to that program. The plan contains:

- Information required by CA Datacom/DB about your program.
- Each embedded SQL statement in your program.

If you modify any SQL statement in your program, you must precompile your program again to update the plan.

**Note:** The Preprocessor options you specify in your host program have no effect on SQL statements submitted through CA Dataquery or the CA Datacom Datadictionary Interactive SQL Service Facility.

## Naming the Plan

When you submit an application with embedded SQL, you can specify the authorization ID (or schema) that owns the application. You use the SQL Preprocessor AUTHID= option to make this assignment.

The value you assign to AUTHID= must be the name of a schema which already exists in the CA Datacom Datadictionary.

In COBOL, you can also specify the plan name for an application using the SQL Preprocessor's PLANAME= option. If you do not assign a value to PLANAME=, the plan name defaults to the value specified for the PROGRAM-ID in your COBOL application.

In PL/I, C, and Assembler, you must also specify the plan name for an application using the SQL Preprocessor's PLANAME= or PLANNAME= option.

In COBOL and Assembler, the plan name must be unique for each program owned by a specific authorization ID.

When you precompile or preprocess a program with embedded SQL, CA Datacom Datadictionary creates a PLAN entity-occurrence and relates this occurrence to the specific program. The name of the PLAN occurrence is formed by concatenating the authorization ID and the plan name, that is to say, the value assigned:

- For COBOL, the PLANAME= option or the PROGRAM-ID.
- For PL/I, C, or Assembler, the PLANAME= or PLANNAME= option.

A plan name remains associated with a specific application. You can update a plan if you make changes to an application by rebinding the plan, or, if necessary, precompiling the application again.

In COBOL, if a rebind fails and you have to precompile the application, you must use the same authorization ID and plan name specified during the first precompile.

## Specifying Processing Options in COBOL

### When to Delete an Existing Plan

If either the authorization ID or plan name is changed in another precompile, the orphan authorization ID and plan name combination needs to be deleted.

### Coding Preprocessor Options in COBOL

To pass options to the Preprocessor, you must code the options on comment lines preceding COBOL's IDENTIFICATION DIVISION statement.

Each comment line specifying Preprocessor options must have an asterisk (\*) in column 7. Immediately following the asterisk, you must enter \$DBSQLOPT as shown in the following example:

```
Input
Column
....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
      *$DBSQLOPT option1=value1 option2=value2 . . .
      IDENTIFICATION DIVISION.
          .
          .
          .
```

When you code Preprocessor options on the comment lines, you can enter the options in any order as long as you adhere to the following rules:

1. Options must be coded between columns 18 and 72, inclusive.
2. No option can be continued from one line to the next, that is to say, the option keyword and assigned value must be coded on the same line.
3. Options must be separated from each other by one or more spaces.
4. Each option you code must be in the form:

```
option=value
```

All options which you enter on the comment lines are edited. The Preprocessor uses the default value for an option in the following situations:

- If you do not specify the option in your program.
- If you enter the option keyword, but do not specify a value after the equal sign.
- If an invalid value is specified for an option.

## Specifying Processing Options in PL/I, C, and Assembler

The three methods by which you can specify processing options are:

- Options file
- Execution parameters
- PL/I, C, or Assembler source

All option specifications are reported in their processing order: first the options file, then the execution parameters, and finally the source.

Not all options can be specified by all three methods. For example, the GENSTOR= and GENINIT= options make sense only within the source. But in all three methods the option specification consists of the option keyword, the equal sign, and the value for the option without any intervening blanks. Case is not important except where noted. See [Options You Can Specify](#) (see page 214) for details about the option keywords.

**Note:** If you are using the C language, specifying GENSTOR=, GENINIT=, or INLINE= is not valid, because in C all storage is allocated at the point of the first EXEC SQL BEGIN DECLARE SECTION, meaning that the *inline* method is the only one C can use.

## Using the Option File Method in PL/I, C, and Assembler

Options specified in the option file are coded one per record beginning in column one. An option specification is considered a comment if it is indented. Anything to the right of the option specification is also considered a comment.

The options file is a good place to put options that are common to a group of programs (or possibly all of the programs at your site).

### PL/I Example

```
//OPTION DD *  
CBSIO=0  
isolevel=c  
  language=pli  
plnclose=t  
PRTY=7  
sqlmode=ansi  
TIMEMIN=0  
msgprec=d  
/*
```

**Note:** The option specification `language=pli` is a comment because it is indented. Also note that case is not important.

### C Example

```
//option dd *  
cbsio=0  
isolevel=c  
  language=c  
plnclose=t  
prty=7  
sqlmode=ansi  
timemin=0  
msgprec=d  
/*
```

**Note:** The option specification `language=c` is a comment because it is indented. Also note that in the C language the case is important, that is to say, using upper case causes errors.

## Assembler Example

```
//OPTION DD *
LANGUAGE=ASM
SQLMODE=DATACOM
CBSIO=0
isoLevel=c
  language=asm
plnclose=t
PRTY=7
sqlmode=ansi
TIMEMIN=0
msgprec=d
/*
```

**Note:** The option specification `language=asm` is a comment because it is indented. Also note that case is not important.

## Using the Execution Parameters Method in PL/I, C, and Assembler

Options specified as execution parameters are separated by commas for z/OS and blanks for z/VSE. This option specification method is useful for options that are unique to a particular program such as the plan name and authorization ID. These options can be specified when executing a procedure by using symbolic substitution.

### Example 1 z/OS for PL/I, C, and Assembler

```
PARM=' PLANNAME=ABCSQL , AUTHID=ADMIN '
```

### Example 2 z/OS for PL/I, C, and Assembler

```
PARM=' PLANNAME=&PGMNM. , AUTHID=&OWNER ' .
```

**Note:** The symbolic substitution for the plan name can also be used to access the correct source member.

### Example 3 z/VSE for PL/I, C, and Assembler

```
PARM=' SMPR=ABC .P ITYP=I '
```

## Using the Source Method in PL/I, C, and Assembler

Option specifications in the source can appear on any record but should not be combined with actual code, because the option specification is commented.

The marker `$DBSQLOPT` is used by the Preprocessor to recognize option specifications. In Assembler and C, start `$DBSQLOPT` in column 1. In PL/I, however, start `$DBSQLOPT` in column 2. In PL/I, C, or Assembler, the `$DBSQLOPT` marker must be followed by at least one blank before the option specifications begin. The marker may be followed by:

- In PL/I and C, one or more specifications separated by commas and terminated with a semicolon.
- In Assembler, one or more specifications separated by commas.

Options of the plan must be specified prior to the first SQL statement in the source because the plan is added at that point.

### Example 1 for PL/I and C

```
$DBSQLOPT authid=sysadm,  
          planname=xyzk  
          ;
```

### Example 1 for Assembler

```
$DBSQLOPT authid=sysadm,          X  
          planname=xyzk          X
```

### Example 2 for PL/I and C

```
$DBSQLOPT authid=sysadm, planname=xyzk ;
```

### Example 2 for Assembler

```
$DBSQLOPT authid=sysadm, planname=xyzk
```

### Example 3 for PL/I and C

```
$DBSQLOPT authid=sysadm,planname=xyzk;
```

### Example 3 for Assembler

```
$DBSQLOPT authid=sysadm,planname=xyzk
```

## Options You Can Specify

This table shows the options you can specify in each language so that the Preprocessor builds a plan for your program.

## Valid Options per Language

<b>COBOL</b>	<b>PL/I</b>	<b>C</b>	<b>Assembler</b>
APOST=			
AUTHID=	AUTHID=	AUTHID=	AUTHID=
CBSIO=	CBSIO=	CBSIO=	CBSIO=
CHECKPLAN=	CHECKPLAN=	CHECKPLAN=	CHECKPLAN=
CHECKWHEN=	CHECKWHEN=	CHECKWHEN=	CHECKWHEN=
CHECKWHO=	CHECKWHO=	CHECKWHO=	CHECKWHO=
COBMODE=			
DATE=	DATE=	DATE=	DATE=
DECPOINT=	DECPOINT= DECPT=	DECPOINT= DECPT=	
GENSECTN=			
	GENSTOR=		GENSTOR=
	GENINIT=		GENINIT=
			INLINE=
ISOLEVEL=	ISOLEVEL=	ISOLEVEL=	ISOLEVEL=
ITYP=	ITYP=	ITYP=	ITYP=
	LANGUAGE= LANG=	LANGUAGE= LANG=	LANGUAGE= LANG=
	MARGINS=		MARGINS=
MSG=			
	MSGEXEC=	MSGEXEC=	MSGEXEC=
	MSGPREC=	MSGPREC=	MSGPREC=
OPT=	OPT=	OPT=	OPT=
PAGESIZE=	PAGESIZE=	PAGESIZE=	PAGESIZE=
PGMNAME=			
PLANAME=	PLANAME= PLANNAME=	PLANAME= PLANNAME=	PLANAME= PLANNAME=
PLNCLOSE=	PLNCLOSE=	PLNCLOSE=	PLNCLOSE=
PRTREXIT=	PRTREXIT=	PRTREXIT=	PRTREXIT=

COBOL	PL/I	C	Assembler
PRTY=	PRTY=	PRTY=	PRTY=
QUOTE=			REFNTRY=
SAVEPLANSEC=	SAVEPLANSEC=	SAVEPLANSEC=	SAVEPLANSEC=
	SMBR=	SMBR=	SMBR=
SQLMODE=	SQLMODE=	SQLMODE=	SQLMODE=
STRDELIM=	STRDELIM= STRDLM= STRINGDELIM=	STRDELIM= STRDLM= STRINGDELIM=	
TIME=	TIME=	TIME=	TIME=
TIMEMIN=	TIMEMIN=	TIMEMIN=	TIMEMIN=
TIMESEC=	TIMESEC=	TIMESEC=	TIMESEC=
	UCRPT=		UCRPT=
USRNTRY=			USRNTRY=
VIEWSEC=	VIEWSEC=	VIEWSEC=	VIEWSEC=
WORKSPACE=	WORKSPACE=	WORKSPACE=	WORKSPACE=

Regarding the CHECKPLAN=, CHECKWHEN=, and CHECKWHO= options that are used in plan security, following is a chart showing which combinations of those options are valid. Reference this chart when studying their descriptions in Description of Options.

Plan Options	Values							
CHECKWHO (B=BINDER, A=ACCESSOR)	B	B	B	B	A	A	A	A
CHECKWHEN (B=BIND, E=EXECUTE)	B	B	E	E	B	B	E	E
CHECKPLAN (N=NO, Y=YES)	N	Y	N	Y	N	Y	N	Y
<b>ALLOWABLE COMBINATION?</b> (Y=YES, 1/2/3 see below)	1	Y	1	2	3	3	Y	Y

**Reason Codes**

1. Not allowed because with plan-level security off, anyone could run this plan, and the executor's table-level privileges would not be checked.
2. Not currently supported.
3. Not allowed because SQL does not know at bind-time whom the executors are.



## Description of Options

### **APOST=**

*(COBOL only.)* Specifies if an apostrophe (') is the delimiting character for character string literals generated in the SQL Communication Area (SQLCA) and the SQL Work Area (SQLWA). This option is provided for compatibility with COBOL compilers which have a similar option.

This option is mutually exclusive with the QUOTE= option, that is to say, if you specify APOST=, do not specify QUOTE= in the Preprocessor options. If neither APOST= or QUOTE= is specified, the Preprocessor uses the default of APOST=Y.

#### **Valid Entries:**

Y (for yes)

#### **Default Value:**

Y for z/OS environment

### **AUTHID=**

Specifies the program plan associated authorization ID.

Any SQL objects (tables, views, synonyms) you create in your program are owned by this authorization ID unless you specifically qualify those objects with a different authorization ID within the program.

The authorization ID name must be 1 to 18 characters.

#### **Valid Entries:**

An authorization ID name of from 1 to 18 characters

#### **Default Value:**

(No default)

### **CBSIO=**

Specifies an I/O limit interrupt value for all SQL commands that create a set. This option allows application environments to establish their own maximums in I/O and set processing relative to their own requirements.

Use this option to limit the computer resources that can be used for each execution of the following statements in the plan:

- OPEN CURSOR, FETCH CURSOR
- SELECT INTO
- INSERT, UPDATE, DELETE
- CREATE INDEX, DROP INDEX, ALTER TABLE

For cursors, the limit applies to the total resources used to OPEN and FETCH all rows of the cursor.

A counter is incremented each time a different index or data block is accessed, and each time 100 rows are read. Execution is terminated, and SQL return code -137 is returned when this counter exceeds the limit.

The value of the counter is reported in the Statistics and Diagnostics Area (PXX) at the end of each request to the Multi-User Facility when any SQL traces are in effect.

For cursor, SELECT INTO, INSERT, UPDATE and DELETE, you can use the total estimated cost reported in the SYSADM.SYSMSG table when bind time optimization messages are requested with the MSG= plan option as a guide for setting the limit. For CREATE INDEX, DROP INDEX, and ALTER TABLE, estimate the limit as the number of bytes in the table divided by 2000. You must set the limit for the most expensive statement in the plan.

A value of 0 (zero) means no limit.

**Note:** Here is how the CBSIO plan option is calculated: to 500,000 is added the amount over 500,000 multiplied by 10,000. For example, given a value of 500,100, the calculation would be  $500,000 + (100 * 10,000) = 500,000 + 1,000,000 = 1,500,000$ .

**Valid Entries:**

0—524286

**Default Value:**

0

**CHECKPLAN=**

This plan option allows the creator of a plan to specify whether that plan is to be secured.

If CHECKPLAN=Y, any accessor ID which attempts to execute the plan must have the PLAN EXECUTE privilege for that plan.

If CHECKPLAN=N, any accessor ID can execute the plan (table-level privileges, however, are still checked).

**Note:** For additional information about plan security, see GRANT and REVOKE. Also see the information on plan security in the *CA Datacom Security Reference Guide*.

See the table of valid combinations of CHECKPLAN=, CHECKWHEN=, and CHECKWHO= presented previously.

**Valid Entries:**

Y or N

**Default Value:**

CHECKPLAN=N is the default only if the CHECKPLAN= parameter in the PLANSEC Multi-User startup option was *not* specified. If the CHECKPLAN= parameter in PLANSEC was specified, its value is the default here. See the *CA Datacom/DB Database and System Administration Guide* for more information about Multi-User startup options.

**CHECKWHEN=**

Specifies whether table-level privileges are to be checked at bind or runtime.

If CHECKWHEN=BIND, then CHECKWHO=BINDER must be specified (it is impossible for SQL to know all potential executors). Similarly, if CHECKWHO=ACCESSOR, then CHECKWHEN=EXECUTE must be specified.

**Note:** For additional information about plan security, see GRANT and REVOKE. Also see the information on plan security in the *CA Datacom Security Reference Guide*.

See the table of valid combinations of CHECKPLAN=, CHECKWHEN=, and CHECKWHO= presented previously.

**Valid Entries:**

BIND or EXECUTE

**Default Value:**

EXECUTE is the default only if the CHECKWHEN= parameter in the PLANSEC Multi-User startup option was *not* specified. If CHECKWHEN= in PLANSEC was specified, its value is the default here. See the *CA Datacom/DB Database and System Administration Guide* for more information about Multi-User startup options.

### CHECKWHO=

Used to specify whether table-level privileges is checked at bind or execute time, and whether the access rights of the binder or the executor are checked. If CHECKWHO=BINDER, the only privilege needed by an accessor ID to run that plan is the PLAN EXECUTE privilege (all table-level privileges required to execute the plan are checked using the binder's accessor-ID). Since the CHECKWHO=BINDER type of plan allows the binder to effectively grant temporary privileges to accessors who use the plan, the ability to create CHECKWHO=BINDER plans must be strictly controlled. To create a CHECKWHO=BINDER plan, you must possess the CHECKBINDER system privilege. See the information about the granting and revoking of the CHECKBINDER system privilege in the *CA Datacom Security Reference Guide*.

Because it is impossible for SQL to know all potential executors, specify CHECKWHO=BINDER if CHECKWHEN=BIND and CHECKWHEN=EXECUTE if CHECKWHO=ACCESSOR.

**Note:** For additional information about plan security, see GRANT and REVOKE. Also see the information on plan security in the *CA Datacom Security Reference Guide*.

See the table of valid combinations of CHECKPLAN=, CHECKWHEN=, and CHECKWHO= on presented previously.

#### Valid Entries:

ACCESSOR or BINDER

#### Default Value:

ACCESSOR is the default only if the CHECKWHO= parameter in the PLANSEC Multi-User startup option was *not* specified. If the CHECKWHO= parameter in PLANSEC was specified, its value is the default here. See the *CA Datacom/DB Database and System Administration Guide* for more information about Multi-User startup options.

### COBMODE=

(*COBOL only.*) Specifies the host language, either OS/VS COBOL or VS/COBOL II.

COBOL II VS/COBOL II Release 3 or later is supported. The CMP2 option is not supported. When using nested programs all SQL statements and any host variables they reference must be within the first program, and all programs must have a DATA DIVISION, a PROCEDURE DIVISION, and a WORKING STORAGE section.

#### Valid Entries:

OSVS, VSCOB2

#### Default Value:

OSVS

**DATE=**

Specifies the DATE output format as follows:

Entry	Format	Description
ISO	yyyy-mm-dd	International Standards Organization
USA	mm/dd/yyyy	IBM USA standard
EUR	dd.mm.yyyy	IBM European standard
JIS	yyyy-mm-dd	Japanese Industrial Standard

**Valid Entries:**

ISO, USA, EUR, JIS

**Default Value:**

The default is the value specified in the Multi-User Facility's DATE startup option.

**Note:** ISO is the default of the Multi-User Facility's DATE startup option.

**DECPOINT=**

(*COBOL, PL/I, and C only.*) Specify C if you want a comma (,) to be the decimal point indicator in decimal, numeric, and floating-point literals.

Specify P if you want a period (.) to be the decimal point indicator.

If the comma is specified as the decimal point indicator, commas which are used as separators must be followed by a space, as in COBOL. Also, any comma followed by a space is interpreted as a separator, even if the comma is preceded by a numeric digit.

**Valid Entries:**

C (for comma)  
P (for period)

**Default Value:**

P

**DECPT=**

(PL/I only.) Same as DECPOINT= (see above).

**GENSECTN=**

(COBOL only.) Specify if you want the Preprocessor to generate COBOL items in the WORKING-STORAGE SECTION or LOCAL-STORAGE SECTION of the program.

If you specify W, the Preprocessor generates the items in the WORKING-STORAGE SECTION of the program.

If you specify O (a letter O, not a zero) for GENSECTN=, data structures (SQLCAs, SQLWAs, SQLDAs), used when SQL statements are executed, are generated in a LOCAL-STORAGE section. Addresses of host variables are stored into these data structures, and VALUE clauses may be used.

Memory for a LOCAL-STORAGE section is allocated and value clauses are executed each time a program is called, as opposed to once per run-unit for WORKING-STORAGE sections. LOCAL-STORAGE therefore provides a way to make programs re-entrant.

When you use a LOCAL-STORAGE section:

- The entire LOCAL-STORAGE SECTION syntax must appear on a single line.
- The same rules apply to LOCAL-STORAGE that apply to WORKING-STORAGE, with respect to required locations relative to other program sections. CA Datacom/DB does not edit with regard to the order of the LOCAL-STORAGE and WORKING-STORAGE sections relative to each other, nor does CA Datacom/DB edit for the number of WORKING-STORAGE or LOCAL-STORAGE sections. The compiler enforces any rules.
- If GENSECTN=O, then a LOCAL-STORAGE section must exist, otherwise an error is generated.

**Valid Entries:**

W (Preprocessor generates COBOL items in the WORKING-STORAGE SECTION)

O (Preprocessor generates COBOL items in the LOCAL-STORAGE SECTION)

**Default Value:**

W

**GENSTOR=**

*(PL/I and Assembler only.)* Causes the necessary storage for SQL to be generated. GENSTOR= is valid only when specified in the source (the \$DBSQLOPT statement).

**Valid Entries:**

TOP specifies the top of the source.  
BOT specifies the end of the source.  
HERE specifies after this point in the source.

**Default Value:**

If not specified, generated storage is placed before the last END statement.

**GENINIT=**

*(PL/I and Assembler only.)* Causes the initialization code to be generated after a specified point in the source. GENINIT= is valid only when specified in the source (the \$DBSQLOPT statement).

**Valid Entries:**

TOP specifies the top of the source.  
BOT specifies the end of the source.  
HERE specifies after this point in the source.

**Default Value:**

If not specified, generated is placed before the last END statement.

**INLINE=**

*(For Assembler)* Specifies the generation of control structures. That is, SQL Work Areas (SQLWAs) that are to be used in conjunction with SQL statements.

With INLINE=N, one set of SQLWAs is generated for each SQL statement. But with INLINE=Y, the Preprocessor generates one set of SQLWAs to use for all SQL statements. INLINE=Y generates more executed code but much less overall code, avoiding the addressability problems (that is, not enough base registers) that can occur when INLINE=N.

**Note:** Because the C Preprocessor uses the inline method exclusively, the C Preprocessor does not look at the INLINE= option. For that reason, an error may occur if you code the INLINE= option when using the C language. Therefore, *when using C, the INLINE= option should not be coded.*

**Valid Entries:**

Y or N

**Default Value:**

N

## ISOLEVEL

Specifies the isolation level, or the degree to which a unit of recovery in your application is isolated from the updating operations of other units of recovery.

If you specify U (for uncommitted data), no locks are acquired for any rows accessed. Your application can access rows that have been updated by another unit of recovery, even though those changes may not have been committed. Since no locks are acquired, no updates, deletes, or inserts may be done by a unit of recovery operating in the U isolation level.

If you specify C (for cursor stability), locks are acquired for all rows accessed. Your application therefore only accesses rows that contain committed data. For updateable cursors (those that have associated UPDATE WHERE CURRENT OF or DELETE WHERE CURRENT OF statements), exclusive locks are acquired. For read-only cursors, share locks are acquired. When a row is fetched, the lock on the previous row of the cursor is released unless the row was modified. All locks are released when the unit of recovery ends, at which time changes are either committed (COMMIT WORK) or rolled back (ROLLBACK WORK).

In C isolation level, the current row of a cursor may or may not be locked while your application is accessing it, depending on whether the cursor is updateable. For updateable cursors, the current row of the cursor is always locked with an exclusive lock when your application fetches it. For read-only cursors, the SQL Facility may read ahead and transfer multiple rows from the Multi-User Facility to your application's region. In this case, the current row of the cursor could possibly no longer be locked when your application fetches it. If your application requires the current row of a cursor to be locked, it must use an updateable cursor to fetch the row.

If you specify R for repeatable read, data once seen by a transaction cannot be changed by another task while the first is still active. Also, other records cannot be added or updated if it would cause them to participate in the set of records seen by the first transaction.

If SQL receives a blank or zero in the ISOLEVEL= specification from any of the SQL access methods, the default for the appropriate SQLMODE is automatically set. If an invalid ISOLEVEL= relative to the effective (that is, specified or defaulted) SQLMODE is explicitly set (nonblank and nonzero), an appropriate SQLCODE and error message is produced.

**Note:** If you specify ANSI or FIPS for the SQLMODE= option, you must also specify ISOLEVEL=C. For more information, see SQLMODE= later in this chapter.

To acquire exclusive control of a table, see [LOCK TABLE](#) (see page 751). Also see [SQL Plan Options Special Topics](#) (see page 270).



**Valid Entries:**

U (No locks are acquired, no changes are allowed)  
 C (Locks are acquired, changes are allowed)  
 R (Locks are acquired, restricted changes are allowed)

**Default Value:**

C (For ANSI and FIPS SQLMODEs)  
 U (For all other SQLMODEs)

**Note:** The *both* parameter of the SQLOPTION Multi-User startup option controls whether you are allowed to mix the use of isolation level U and C plans under a single logical unit of work (LUW) for those LUWs not running any SQLMODE ANSI or FIPS plans. YES specified for the *both* option of SQLOPTION indicates that mixing is allowed.

**ITYP=**

(*z/VSE only*) Specifies an optional file type for SQL INCLUDE files in z/VSE. The type is one letter.

**Note:** In COBOL, the ITYP= specification can be overridden by explicitly coding a file type on the INCLUDE statement.

**Valid Entries:**

Any letter (including, although not recommended, the standard default letters listed below)

**Default Value:**

C for COBOL, P for PL/I and Assembler, H for the C language

**LANGUAGE= or LANG=**

(*PL/I, C, and Assembler only.*) Specifies the source language to be processed. This option overrides the initial values for MARGINS= established at initialization.

**Note:** Specifying MARGINS= is not allowed if you are using the C language.

**Valid Entries:**

PLI for PL/I  
 C for the C language  
 ASM or ASSEMBLER for Assembler

**Default Value:**

PLI for PL/I  
 No default for the C language  
 No default for Assembler

### **MARGINS=**

*(PL/I and Assembler only.)* Specifies the valid columns of the source record inclusively.

Do not specify MARGINS= in the \$DBSQLOPT statement (source code). Specify MARGINS= in the OPTIONS file or in the PARM= text on the EXEC statement of the JCL.

In PL/I you can specify (start, end) or (left, right). For example: (s) or (s,e) or (,e). In Assembler, you can specify (start, end, continue) or (left, right, continue). For example: (s) or (s,e) or (,e) or (s,e,c).

Values are merged with defaults, that is to say, if only one value is specified, the default is assigned to the nonspecified value.

An example in PL/I: if (,71) is specified after LANGUAGE=, the result is (2,71) because 2 is the default for the start value. Or, if (5) is specified, the result is (5,72) because 72 is the default for the end value.

An example in Assembler: if (,71) is specified, the result is (1,71) because 1 is the default for the start value. Or, if (5) is specified, the result is (5,71) because 71 is the default for the end value. The continuation field is the next column after the end (right) margin.

#### **Valid Entries:**

1—30, s < e, and width >= 70

#### **Default Value:**

PL/I defaults are:

1 (start) 72 (end) if specified *before* LANGUAGE=

2 (start) 72 (end) if specified *after* LANGUAGE=

Assembler defaults are:

1 (start)

71 (end)

16 (continue)

### **MSG=**

*(COBOL only.)* Specifies the level of messages you wish the SQL Optimizer to generate. Specify the optimization message in groups of two letters, for example MSG=xy, where S and D and N replace the x and y, and where:

- x refers to precompile-time messages (these are included at the end of the Preprocessor Source Listing), and
- y refers to messages generated by the Optimizer when the statement is executed (these messages may be retrieved from the SYSMMSG table after the statement has been executed).

See SQL Query Optimization Messages for a description of the SYSMMSG table.

**Note:** Messages for a plan are deleted when the plan is deleted. When you re-preprocess a program, the Preprocessor deletes the previous plan and therefore also its diagnostic messages.

Valid combinations of S and D and N are given in Valid Entries below, where:

- S specifies summary
- D specifies detail
- N specifies none

**Valid Entries:**

SS, DD, SD, DS, NS, ND, DN, SN, NN

**Default Value:**

NN

**MSGEXEC=**

*(PL/I, C, and Assembler only.)* Refers to messages generated by the Optimizer when the statement is executed (these messages may be retrieved from the SYSMSG table after the statement has been executed). See SQL Query Optimization Messages for a description of the SYSMSG table.

- S specifies summary
- D specifies detail
- N specifies none

**Note:** Messages for a plan are deleted when the plan is deleted. When you re-preprocess a program, the Preprocessor deletes the previous plan and therefore also its diagnostic messages.

**Valid Entries:**

S, D, N

**Default Value:**

N

**MSGPREC=**

(*PL/I, C, and Assembler only.*) Refers to precompile-time messages. These messages are included at the end of the Preprocessor Source Listing.

- S specifies summary
- D specifies detail
- N specifies none

**Valid Entries:**

S, D, N

**Default Value:**

N

**OPT=**

Specifies the join optimization mode. P specifies normal join optimization. Specify M (manual join order) if the normal join optimization is unacceptable and you want tables joined as they are listed in the FROM clause. This results in a nested loop join.

Do not specify E; it is reserved for future use.

**Valid Entries:**

P or M

**Default Value:**

P

**PAGESIZE=**

Specifies the number of output lines per page on SYSPRINT.

**Valid Entries:**

For COBOL: 0—120

For PL/I, C, and Assembler: 10—255

**Default Value:**

55

**PGMNAME=**

This option enables the CA Datacom Datadictionary name of a program to be changed without requiring the program source itself to be altered. The program name of a procedure must match its external (or load module) name.

The syntax is as follows: PGMNAME=name

The name specified must be a CA Datacom Datadictionary entity-occurrence name. It overrides any PROGRAM-ID specified in COBOL. If the program is a procedure, this name must match both the generated load module name and the EXTERNAL NAME specified in the CREATE PROCEDURE statement.

**Valid Entries:**

A valid name as described above

**Default Value:**

(No default)

**PLANAME=**

Specifies the name for your plan, a name that should be unique within your authorization ID. You can use MIXED strings in a plan name, that is, strings in which both Double-Byte Character Set (DBCS) and Single-Byte Character Set (SBCS) characters are used.

The plan name must be 1 to 18 bytes in length. The first character of the plan name must be alphabetic (including Katakana symbols) or a Shift-Out character (when you are using MIXED strings). Shift-Out and Shift-In characters (used to delimit DBCS substrings) count toward the 18-byte length limit. If you specify more than 18 bytes for the name, the Preprocessor truncates your entry to the first 18 bytes.

For more information on MIXED data and MIXED strings, see [Character Strings](#) (see page 495).

If you specify a name you used previously for a plan, you are, in essence, replacing that existing plan with a new plan. For example, you have an existing plan named PAYROLL. If you specify PAYROLL as the name of a new plan, this new plan replaces the previous plan named PAYROLL.

If you do not specify a plan name, the Preprocessor uses the PROGRAM-ID specified in your program.

**Valid Entries:**

A name 1 to 18 bytes long, first character alphabetic (including Katakana symbols) or a Shift-Out character (when you are using MIXED strings)

**Default Value:**

In COBOL and PL/I, there is no default.

#### Plan Versioning - COBOL Only:

You can generate plans that include a date/timestamp, YYMMDDHHMM

- PLANAME=@TIMESTAMP

Plan name = *program id*YYMMDDHHMM

- If the *program id* is less than 8 bytes, fill up to 8 bytes with '\_'s
- If the *program id* is greater than 8 bytes, truncate it to 8 and concatenate the timestamp. You receive warning message DB21013W.
- Example for PROGRAM-ID.CDC100 compiled on 2015/04/15 at 11:45 am:

CDC100\_\_1504171145

- PLANAME=*value*@TIMESTAMP

Plan name = *value*YYMMDDHHMM

- If *value* is less than 8 bytes, fill up to 8 bytes with '\_'s
- Example for PROGRAM-ID.CDC100 compiled on 2015/04/17 at 2:07 pm, but with PLANAME=CDC100A@TIMESTAMP

CDC100A\_1504171407

- Since @TIMESTAMP is 10 bytes, the same as the generated timestamp, the value in PLANAME cannot be greater than 8 characters.
- Plans are not overwritten unless they are generated within the same minute. If so, the last one remains.
- In the listing for the COBOL program, the plan name is displayed as SQLCA-PLAN-NAME within the SQLCA block of the WORKING-STORAGE SECTION.
- As part of this process, a flag byte is set to x'02' in the SQL options block. This is also stored in the DDD. It can be accessed to find just those plans that have been added to the system using the @TIMESTAMP option. For two variations of obtaining this information, see [Example SQL Statements](#) (see page 135).

**PLANNAME=**

(PL/I and Assembler only.) Same as PLANAME= (see previous description).

**PLNCLOSE=**

Specifies when the plan, and any User Requirements Tables automatically opened by the SQL Manager, are closed.

If you specify T, the plan, and any User Requirements Tables automatically opened by the SQL Manager, close when the transaction ends, that is to say, an SQL COMMIT WORK or ROLLBACK WORK statement, a CA Datacom/DB LOGCP, LOGCR or LOGTB command, or a CA Datacom CICS Services DEQUE.

We recommend the T option for a CICS environment. We also recommend PLNCLOSE=T for procedures. We recommend the R option for batch programs.

If you specify R, the plan, and any User Requirements Tables automatically opened by the SQL Manager, close when the run unit ends, or when a CA Datacom/DB CLOSE command is issued. In a CA Datacom CICS Services environment, the run unit ends only when CICS is terminated or when the SQL User Requirements Table (usually URT 20) is closed using the DBOC command. When ISOLEVEL=U is used, because no locks are acquired by the MUF (and therefore no DEQUE commands are issued by CA Datacom CICS Services), the plan and User Requirements Tables are not closed at the end of each CICS transaction, even if PLNCLOSE=T. A PLNCLOSE=R plan can be preprocessed or rebound with DDOL or DBSRFPR before the run unit ends if no current unit of recovery has executed the plan.

User Requirements Tables opened on behalf of a plan with PLNCLOSE=R are not closed until the plan closes when the SQL=YES User Requirements Table (default User Requirements Table 020) is closed. If User Requirements Tables accessing a database need to be closed to perform utility functions, you can close those SQL-generated User Requirements Tables accessing a database by deleting (if you have the *delete* privilege) those rows in the SQL\_STATUS\_URT\_INACTIVE *virtual table* (this *virtual table* is a view on the SQL\_STATUS\_URT table with the restriction that only User Requirements Tables for the current run unit with zero users are selected). Following is an example query that can be executed from any tool that uses SQL, where nnn is the database-ID to be closed. Any valid WHERE clause can be used, including no WHERE clause to close all User Requirements Tables.

```
DELETE FROM SYSADM.SQL_STATUS_URT_INACTIVE
WHERE DBID = 'nnn';
```

Closing these User Requirements Tables does not keep them from being reopened. There can also be other User Requirements Tables that are active, meaning they have one or more plans that have accessed the table in their current transaction. In addition, other run units can also have User Requirements Tables open for the database. Use the following query to see which User Requirements Tables are open for a database:

```
SELECT * FROM SYSADM.SQL_STATUS_URT
WHERE DBID = 'nnn';
```

For information about possible performance enhancement using the Least Recently Used (LRU) statement cache to disconnect the caching of plan statements from the control of the PLNCLOSE= option, see **LRU Statement Cache**.

**Valid Entries:**

T (close when transaction ends)  
R (close when run unit ends)

**Default Value:**

R

**PROCSQLUSAGE=**

Is valid for COBOL, PL/I, Assembler, and C. If this option is specified, SQL prepares the program for execution as a procedure. This option is required for programs intended to run as procedures. It is prohibited for programs not run as procedures.

**Note:** All procedures must be preprocessed, even if there are no embedded SQL statements in a particular procedure.

**Important!** Use PROCSQLUSAGE= only as documented. If it is specified or omitted inappropriately, the program can fail in an unpredictable way before SQL has a chance to detect the error.

To support procedure execution, the preprocessors add code to programs that specify PROCSQLUSAGE=, thus making proper use of the option critical. Therefore, make certain you have coded this option accurately. CA Datacom reserves the right to add edits at any time.

Specifying NO means the procedure does not call CA Datacom SQL.

Specifying CONTAINS means that the procedure calls CA Datacom SQL but contains no SELECT, SELECT INTO, preparations of dynamic-select, INSERT, UPDATE, or DELETE statements.

Specifying READS means that the procedure contains a SELECT, SELECT INTO, or preparation of a dynamic-select statement, but does not contain INSERT, UPDATE, or DELETE statements.



Specifying MODIFIES means that the procedure contains at least one INSERT, UPDATE, or DELETE statement.

CA Datacom only checks for the existence of calls to CA Datacom SQL in the procedure at the time of the CREATE PROCEDURE statement. (An error is produced, however, if NO is specified in a program containing SQL, or vice versa.) CA Datacom requires the value specified to match the corresponding specification in the CREATE PROCEDURE statement.

**Valid Entries:**

NO, CONTAINS, READS, or MODIFIES

**Default Value:**

(No default)

**PRTREXIT=load-module-name**

Specifying PRTREXIT= allows you to write a printer exit routine to print the output instead of allowing the Preprocessor to write to SYSPRINT. The load-module-name is the name of your printer exit routine. When your printer exit routine is called, the registers are as follows:

**Register 1**

Address of parameter list as follows:

Word 1 = x'00000014'

Word 2 = AL1(length of print line)

AL3(address of print line)

Word 3 = Address of a 1-byte top-of-page indicator:

if bit x'20' is on, top-of-page is requested

**Register 13**

Address of a register save area which you must use to save and restore the CA Datacom/DB registers according to standard linkage conventions.

**Register 14**

Address to return to inside CA Datacom/DB.

**Register 15**

Address of the entry point of your printer exit.

On return from your routine, the contents of all registers (except 15) should contain what they contained before the exit was called. Register 15 should contain 0 unless a failure occurred. The Preprocessor aborts processing if a nonzero register 15 is returned.

Remember to concatenate the library containing your printer exit to the end of the Preprocessor load library concatenation in your JCL.

If you do not specify a load-module-name (with PRTREXIT=), processing continues to write to SYSPRINT instead of calling the printer exit.

**Valid Entries:**

A load-module-name of up to eight characters

**Default Value:**

(No default)

**PRTY=**

Specifies the priority of the SQL requests from the plan within the Multi-User Facility. The lowest priority is 1, while 15 is the highest priority.

If you need more information about specifying a priority, see your Database Administrator.

**Valid Entries:**

1—15

**Default Value:**

7

**QUOTE=**

*(COBOL only.)* Specifies if a quotation mark (") is the delimiting character for character literals generated in the SQL Communication Area (SQLCA) and the SQL Work Area (SQLWA). This option is provided for compatibility with COBOL compilers which have a similar option.

This option is mutually exclusive with the APOST= option, that is to say, if you specify QUOTE=, do not specify APOST= in the Preprocessor options.

If neither QUOTE= or APOST= is specified, the Preprocessor uses the default of QUOTE=Y for z/VSE environments.

**Valid Entries:**

Y (for yes)

**Default Value:**

Y for z/VSE environments

**REFNTRY=**

*(Assembler only.)* See USRNTRY= later in this chapter.

**SAVEPLANSEC=**

Use this option to specify whether to drop or not to drop security privileges granted on a PLAN when a program is re-preprocessed.

SAVEPLANSEC=Y means PLAN privileges are not dropped and therefore do not have to be regranted after re-preprocessing a program.

SAVEPLANSEC=N means PLAN privileges are dropped (revoked).

**Valid Entries:**

Y or N

**Default Value:**

N

**SMBR=**

*(PL/I, C, and Assembler only.)* Specifies the member name and type for the source residing in a z/VSE library. The value specified must be of the form: name.type (for example, SMBR=ABC.P where ABC is the name and P the type).

If SMBR= is not specified, the source is assumed to be on sequential disk. If you specify SMBR=, you must do so either in the execution parameters or the options file. If you specify SMBR= when the source is on a sequential disk file, an open error results.

**Valid Entries:**

A valid member name and a type of length 1

**Default Value:**

(No default)

**SQLMODE=**

Specifies the mode in which to process the program. If you specify SQLMODE=DATACOM, your program is processed in extended mode, which means CA Datacom/DB extensions to the standards are allowed in your SQL statements. Names for tables, columns, views, synonyms and cursors can be 1 to 32 characters in length if SQLMODE=DATACOM. Authorization IDs and plan names must be 1 to 18 characters in extended mode.

Specifying SQLMODE=DB2 allows you to use the CA Datacom/DB DB2 compatibility mode. CA Datacom DB2 Transparency is required to use the CA Datacom/DB DB2 mode. The DB2 compatibility mode allows you to use application programs written for IBM DB2. CA Datacom/DB recompiles and executes DB2 application programs against CA Datacom/DB tables without your having to change the source code of those programs. Plans created by the CA Datacom DB2 Transparency Bind program also execute in DB2 mode.

In COBOL and PL/I you can specify `SQLMODE=DB2A86` to use the CA Datacom/DB2 compatibility mode while conforming to ANSI 86 standards. In COBOL and PL/I, specify ANSI or FIPS for your program to be processed in ANSI or FIPS mode, which means all your SQL statements must be coded according to ANSI or FIPS standards. When ANSI or FIPS mode is specified, the `ISOLEVEL=U` option is not allowed. `ISOLEVEL=C` must be specified when `SQLMODE=ANSI` or `SQLMODE=FIPS`. Authorization IDs and plan names must be 1 to 18 characters in ANSI mode. Names for tables, columns, views, synonyms and cursors must be 1 to 18 characters in length if `SQLMODE=ANSI` or `SQLMODE=FIPS`.

**Note:** The `SQLMODE` Multi-User Facility startup option must be set to `DATACOM` before this Preprocessor option is effective. If the `SQLMODE` Multi-User Facility startup option is set to `ANSI` or `FIPS`, this Preprocessor option is overridden and all SQL statements must comply with ANSI or FIPS standards. See your Database Administrator for information on the value assigned to the `SQLMODE` Multi-User Facility startup option.

**Valid Entries:**

For COBOL and PL/I:  
DATACOM, DB2, DB2A86, ANSI, FIPS  
For Assembler: DATACOM, DB2

**Default Value:**

DATACOM

**STRDELIM=**

*(COBOL, PL/I, and C only.)* Specifies whether you want the string delimiter, used to delimit character string literals in SQL statements, to be an apostrophe (') or a quotation mark (").

The escape character, used to enclose delimited SQL identifiers, is the apostrophe if the string delimiter is the quotation mark, or the quotation mark if the string delimiter is the apostrophe. See *Delimited SQL identifiers* for more information on delimited SQL identifiers.

Specify A for apostrophe or Q for quotation mark.

**Valid Entries:**

A, Q

**Default Value:**

A

**STRDLM=**

(PL/I only.) Same as STRDELIM= (see previous description).

**STRINGDELIM=**

(PL/I only.) Same as STRDELIM= (see previous description).

**TIME=**

Specifies the TIME output format as follows:

Entry	Format	Description
ISO	hh.mm.ss	International Standards organization
USA	hh:mm AM or PM	IBM USA standard
EUR	hh.mm.ss	IBM European standard
JIS	hh:mm:ss	Japanese Industrial Standard

**Valid Entries:**

ISO, USA, EUR, JIS

**Default Value:**

The default is the value specified in the Multi-User Facility's TIME startup option.

**Note:** ISO is the default of the Multi-User Facility's TIME startup option.

**TIMEMIN=**

Specifies exclusive control wait time limit in minutes.

This option allows a program to either wait or not wait for an explicit amount of time when another job is holding a requested record under exclusive control. If the specified time is exceeded, the application program receives a -117 value in the SQLCODE of the SQL Communication Area and a CA Datacom/DB 61 return code to inform the user that the record was not available.

Specifying a zero for both TIMEMIN= and TIMESEC= means that there is no time limit, and without a limit on the wait time, a *wait forever* condition is possible.

TIMEMIN=0 and TIMESEC=1 means do not wait at all. *Do not specify nonzero values for both TIMEMIN= and TIMESEC=.*

If you are using CA Datacom STAR for distributed processing, see CA Datacom STAR documentation before specifying this option.

**Valid Entries:**

0—120

**Default Value:**

0

**TIMESEC=**

Specifies exclusive control wait time limit in seconds.

This option allows a program to either wait or not wait for an explicit amount of time when another job is holding a requested record under exclusive control. If the specified time is exceeded, the application program receives a -117 value in the SQLCODE of the SQL Communication Area and a CA Datacom/DB 61 return code to inform the user that the record was not available.

Specifying a zero for both TIMEMIN= and TIMESEC= means that there is no time limit, and without a limit on the wait time, a *wait forever* condition is possible.

TIMESEC=1 and TIMEMIN=0 means do not wait at all. *Do not specify nonzero values for both TIMEMIN= and TIMESEC=.*

If you are using CA Datacom STAR for distributed processing, see CA Datacom STAR documentation before specifying this option.

**Valid Entries:**

0—120

**Default Value:**

0

**UCRPT=**

*(PL/I and Assembler only.)* Specifies whether report should be uppercase only. The data is not affected.

Do not specify UCRPT= in the \$DBSQLOPT statement (source code). Specify UCRPT= in the OPTIONS file or in the PARM= text on the EXEC statement of the JCL. In PL/I, if this is the first option in the execution parameters, the entire report is in uppercase.

In Assembler, if this is the first option in the option file, the report is uppercase, except for the initial title lines. To have everything in uppercase, the option should be coded in the execution parameters.

**Valid Entries:**

Y, N

**Default Value:**

N

**USRNTRY=**

(*COBOL and Assembler only.*) The description of USRNTRY= differs in COBOL and Assembler.

**COBOL Description:**

Use USRNTRY= in COBOL to specify the entry point in the COBOL program. The value you assign this option **must match** the value specified for the USRNTRY= parameter of the User Requirements Table.

The default is DBMSCBL. If you have changed this in the User Requirements Table, you must enter the same entry point name as specified in the User Requirements Table. Specify NONE if no entry point is to be generated (used when an SQL program is called by another program).

**Valid Entries:** DBMSCBL or

An entry point name consistent with COBOL naming conventions, or

NONE

**Default Value:** DBMSCBL

**Assembler Description:**

USRNTRY= works with the REFENTRY= Preprocessor option to cause generation of an entry point that allows use of a single User Requirements Table with OPEN=DB for many separate programs. USRNTRY= and REFENTRY= are for Assembler batch mode only. Following is shown the generation:

```
usrntry-name EQU    refntry-name
                ENTRY  usrntry-name
```

Use USRNTRY= to specify the name of the generated entry point. USRNTRY= may be entered in two ways. If USRNTRY=NONE, no entry point is generated. If any other value is entered, that value is considered a valid name. This name is also the name in the User Requirements Table USRNTRY= operand. If USRNTRY= is not specified, then a default name, SQLEXECE, may be used to generate the entry point.

REFENTRY= is the name of a CSECT or ENTRY in the program being processed. This name represents the point where the program would get control when called.

In DATACOM mode, USRNTY= and REFNTY= are not required. If REFNTY= is coded, the entry point is generated. If USRNTY= is not coded when REFNTY= is, the default name is used. Specifying USRNTY= without REFNTY= causes an error.

In DB2 mode, the entry point generation must be specified or explicitly suppressed. To specify generation, REFNTY= must be coded, but in this case USRNTY= is optional since the default name, SQLECE, is taken. To suppress generation, USRNTY=NONE must be entered.

**Valid Entries:** A name of up to 8 characters or NONE for USRNTY=, or for REFNTY= a name of up to 8 characters

**Default Value:** SQLECE is the default for USRNTY=, but for REFNTY= there is no default

#### **VIEWSEC=**

Whether view security is used for a particular plan is based on the value of the VIEWSEC= Preprocessor plan option. If VIEWSEC= is not specified, whether a plan uses view security is determined by the value of the view-security specification in the SQLOPTION Multi-User startup option. If neither VIEWSEC= nor the view-security specification in SQLOPTION is used, view security is not used for newly bound or rebound plans.

Specify Y to indicate that view security is to be used during the execution of newly prepared and newly rebound plans.

Specify N to indicate that view security is not to be used during the execution of newly prepared and newly rebound plans.

**Note:** The default for the VIEWSEC= Preprocessor option is the value of the view-security option in the SQLOPTION Multi-User startup option (see the *CA Datacom/DB Database and System Administration Guide* for more information on SQLOPTION) or N if no default was specified.

Also note, the choice of security method is made at prepare-time rather than during execution. A choice of Y is rejected if view security has not been activated for the Multi-User Facility using external security. See the *CA Datacom Security Reference Guide* for more information.

**Valid Entries:**

Y or N

**Default Value:**

Value of the view-security specification in the SQLOPTION Multi-User startup option, which itself defaults to N



**Important!** Subsequently rebound plans (rebound explicitly or automatically) that do not have an explicit view security specification are caused by the value of the SQLOPTION view-security option to change security methods, if necessary, to match the specification. Be aware, therefore, that the security method used by existing plans can be changed intentionally or inadvertently in this way.

#### **WORKSPACE=**

**Use WORKSPACE= only at the direction of CA Support.**

This option specifies an increase in the amount of workspace used at plan execution time. The default is 0 if not specified or an incorrect value is given.

#### **Valid Entries:**

0 to 128

#### **Default Value:**

0

## COBOL Examples

The following show how different Preprocessor options can be coded in COBOL.

### COBOL Example 1

```
Input
CoLumn
....+....1....+....2....+....3....+....4....+....5....+....6....+....7..

*$DBSQLOPT AUTHID=JONES PLANAME=KOLLARC CBSIO=25000 PRTY=7
*$DBSQLOPT SQLMODE=ANSI TIMEMIN=10 TIMESEC=0 PLNCLOSE=T PAGESIZE=88
IDENTIFICATION DIVISION.
      .
      .
      .
```

### COBOL Example 2

```
Input
CoLumn
....+....1....+....2....+....3....+....4....+....5....+....6....+....7..

*$DBSQLOPT AUTHID=JONES CBSIO=30000 PRTY=9 SQLMODE=DATACOM
*$DBSQLOPT TIMEMIN=10 TIMESEC=0 PLNCLOSE=R ISOLEVEL=C
IDENTIFICATION DIVISION.
      .
      .
      .
```

## SQL Communication Area (SQLCA)

The CA Datacom/DB SQL Preprocessor generates one SQLCA for each compiled embedded SQL program. The SQLCA is a collection of variables used by CA Datacom/DB to provide an application program with information about the execution of its SQL statements. Since CA Datacom/DB updates the SQLCA during the execution of every SQL statement, the information in this area applies to the most recently executed SQL statement.

### SQLCA in COBOL

With the CA Datacom/DB SQL Preprocessor for COBOL, the possible formats of the SQLCA are:

- CA Datacom/DB format (see [SQLCA - CA Datacom/DB Format \(COBOL\)](#) (see page 244)). For a table containing descriptions of this example, see [Description of SQLCA in CA Datacom/DB Format](#) (see page 251).
- DB2 format (see [SQLCA - DB2 Format \(COBOL\)](#) (see page 256)). For a table containing descriptions of this example, see [Description of SQLCA in DB2 Format](#) (see page 259).

In COBOL, the Preprocessor always generates the SQLCA structure. An INCLUDE SQLCA is not required. The Preprocessor ignores the INCLUDE directive if SQLCA is the member name. You may therefore either explicitly code the include of the SQLCA or omit it.

In DB2 mode, the CA Datacom/DB SQL Communication Area (SQLCA) used is the DB2 SQLCA, including the values for SQLCODE and SQLERRM.

### SQLCA in PL/I

With the CA Datacom/DB SQL Preprocessor for PL/I the possible formats of the SQLCA are:

- CA Datacom/DB format for ANSI (see [SQLCA - CA Datacom/DB Format \(PL/I\)](#) (see page 246)) and for non-ANSI (see [SQLCA - CA Datacom/DB Format \(PL/I\)](#) (see page 246)). For a table containing descriptions of these examples, see [Description of SQLCA in CA Datacom/DB Format](#) (see page 251).
- DB2 format for ANSI (see [SQLCA - DB2 Format \(PL/I\)](#) (see page 256)) and for non-ANSI (see [DB2 format \(non-ANSI\)](#) (see page 257)). For a table containing descriptions of these examples, see [Description of SQLCA in DB2 Format](#) (see page 259).

In DB2 mode in PL/I, the CA Datacom/DB SQL Communication Area (SQLCA) used is the DB2 SQLCA, including the values for SQLCODE (for the non-ANSI version) or SQLCADE (for the ANSI 86 version), and SQLERRM.

## SQLCA in Assembler

With the CA Datacom/DB SQL preprocessor for Assembler, the possible formats of the SQLCA are:

- CA Datacom/DB format (see [SQLCA - CA Datacom/DB Format \(Assembler\)](#) (see page 248)). For a table containing descriptions of this example, see [Description of SQLCA in CA Datacom/DB Format](#) (see page 251).
- DB2 format (see [SQLCA - DB2 Format \(Assembler\)](#) (see page 258)). For a table containing descriptions of this example, see [Description of SQLCA in DB2 Format](#) (see page 259).

In DB2 mode in Assembler, the CA Datacom/DB SQL Communication Area (SQLCA) used is the DB2 SQLCA, including the values for SQLCODE and SQLERRM.

In Assembler, the SQLCA can be generated in an area separate from the SQLDSECT by entering this line in the source (if this include is not used, the SQLCA is generated in the SQLDSECT by default):

```
EXEC SQL INCLUDE SQLCA
```

## SQLCA in C Language

In the C language there is a single conditional structure for the SQL Communication Area (SQLCA). This single structure generates the correct format for DB2 and ANSI based on the environment in which it is compiled. See [SQLCA - \(C Language\)](#) (see page 249) for an example of the SQLCA in C.

## Example SQLCA Formats

These SQLCA examples follow:

- COBOL (CA Datacom) - see [SQLCA - CA Datacom/DB Format \(COBOL\)](#) (see page 244)
- PL/I (CA Datacom) - see [SQLCA - CA Datacom/DB Format \(PL/I\)](#) (see page 246)
- Assembler (CA Datacom) - see [SQLCA - CA Datacom/DB Format \(Assembler\)](#) (see page 248)
- C (CA Datacom) - see [SQLCA - \(C Language\)](#) (see page 249)
- Description (CA Datacom) - see [Description of SQLCA in CA Datacom/DB Format](#) (see page 251)
- COBOL (DB2) - see [SQLCA - DB2 Format \(COBOL\)](#) (see page 256)
- PL/I (DB2) - see [SQLCA - DB2 Format \(PL/I\)](#) (see page 256)
- Assembler (DB2) - see [SQLCA - DB2 Format \(Assembler\)](#) (see page 258)

- C (DB2) - see [SQLCA - \(C Language\)](#) (see page 259)
- Description (DB2) - see [Description of SQLCA in DB2 Format](#) (see page 259)

### SQLCA - CA Datacom/DB Format (COBOL)

```
01 SQLCA.  
  05 SQLCA-EYE-CATCH          PIC X(08).  
  05 SQLCAID REDEFINES SQLCA-EYE-CATCH  
                                PIC X(08).  
  05 SQLCA-LEN                PIC S9(9) COMP.  
  05 SQLCABC REDEFINES SQLCA-LEN  
                                PIC S9(9) COMP.  
  05 SQLCA-DB-VRS             PIC X(02).  
  05 SQLCA-DB-RLS             PIC X(02).  
  05 SQLCA-LUWID              PIC X(08).  
  05 SQLCA-SQLCODE            PIC S9(9) COMP.  
  05 SQLCA-ERROR-INFO.  
    10 SQLCA-ERR-LEN          PIC S9(4) COMP.  
    10 SQLCA-ERR-MSG          PIC X(80).  
  05 SQLERRM REDEFINES SQLCA-ERROR-INFO.  
    10 SQLERRML               PIC S9(4) COMP.  
    10 SQLERRMC               PIC X(70).  
    10 SQLERRMF               PIC X(10).  
  05 SQLCA-ERROR-PGM          PIC X(08).  
  05 SQLERRP REDEFINES SQLCA-ERROR-PGM  
                                PIC X(08).  
  05 SQLCA-FILLER-1           PIC X(02).  
  05 SQLCA-ERROR-DATA.
```

```
10 SQLCA-DSFCODE      PIC X(04).
10 SQLCA-INFCODE      PIC S9(9) COMP.
10 SQLCA-DBC CODE.
   15 SQLCA-DBC CODE-EXT PIC X(02).
   15 SQLCA-DBC CODE-INT PIC S9(4) COMP.
10 SQLCA-MISC-CODE1   PIC S9(9) COMP.
10 SQLCA-MISC-CODES-B.
   15 SQLCA-MISC-CODE2 PIC S9(9) COMP.
   15 SQLCA-MISC-CODE3 PIC S9(9) COMP.
10 SQLCA-ERR-INFO-2  REDEFINES SQLCA-MISC-CODES-B.
   15 SQLCA-SQLSTATE  PIC X(05).
   15 SQLCA-FILLER-2  PIC X(03).
05 SQLCA-WRN-AREA.
   10 SQLCA-WARNING   PIC X OCCURS 8 TIMES.
05 SQLWARN REDEFINES SQLCA-WRN-AREA.
   10 SQLWARN0        PIC X.
   10 SQLWARN1        PIC X.
   10 SQLWARN2        PIC X.
   10 SQLWARN3        PIC X.
   10 SQLWARN4        PIC X.
   10 SQLWARN5        PIC X.
   10 SQLWARN6        PIC X.
   10 SQLWARN7        PIC X.
05 SQLCA-PGM-NAME    PIC X(08).
05 SQLCA-AUTHID      PIC X(18).
05 SQLCA-PLAN-NAME   PIC X(18).
```

**Note:** All REDEFINES are for compatibility with other SQL implementations.

## SQLCA - CA Datacom/DB Format (PL/I)

```

DCL 1 SQLCA,
    5 SQLCA_EYE_CATCH      CHAR(8) INIT('SQLCA***'),
    5 SQLCA_LEN            FIXED BINARY(31) INIT(196),
    5 SQLCA_DB_VRS        CHAR(2) INIT('08'),
    5 SQLCA_DB_RLS        CHAR(2) INIT('10'),
    5 SQLCA_LUWID         CHAR(8) INIT(' '),
    5 SQLCA_CODE          FIXED BINARY(31),
    5 SQLCA_ERR_LEN       FIXED BINARY(15),
    5 SQLCA_ERR_MSG       CHAR(80) INIT(' '),
    5 SQLCA_ERROR_PGM     CHAR(8) INIT(' '),
    5 SQLCA_FILLER_1      CHAR(2) INIT(' '),
    5 SQLCA_DSFCODE       CHAR(4) INIT(' '),
    5 SQLCA_INFCD         FIXED BINARY(31),
    5 SQLCA_DBCODE_EXT    CHAR(2) INIT(' '),
    5 SQLCA_DBCODE_INT    FIXED BINARY(15),
    5 SQLCA_MISC_CODE1    FIXED BINARY(31),
    5 SQLCA_MISC_CODE2    FIXED BINARY(31),
    5 SQLCA_MISC_CODE3    FIXED BINARY(31),
    5 SQLCA_WRN_AREA,
    10 SQLCA_WARNING (0:7) CHAR(1) INIT(' '),
    5 SQLCA_PGM_NAME      CHAR(8) INIT(' '),
    5 SQLCA_AUTHID        CHAR(18) INIT('authid here '),
    5 SQLCA_PLAN_NAME     CHAR(18) INIT('plan name here ');
DCL SQLCAID              CHAR(8)
    DEFINED SQLCA_EYE_CATCH;
DCL SQLCABC              FIXED BINARY(31)
    DEFINED SQLCA_LEN;
DCL SQLERRML            FIXED BINARY(15)
    DEFINED SQLCA_ERR_LEN;
DCL SQLERRMC            CHAR(80)
    DEFINED SQLCA_ERR_MSG;
DCL SQLERRP             CHAR(8)
    DEFINED SQLCA_ERROR_PGM;
DCL 1 SQLWARN
    DEFINED SQLCA_WRN_AREA,
    5 SQLWARN0           CHAR(1),
    5 SQLWARN1           CHAR(1),
    5 SQLWARN2           CHAR(1),
    5 SQLWARN3           CHAR(1),
    5 SQLWARN4           CHAR(1),
    5 SQLWARN5           CHAR(1),
    5 SQLWARN6           CHAR(1),
    5 SQLWARN7           CHAR(1);

```

```

DCL 1 SQLCA_WARN          DEFINED SQLCA_WRN_AREA,
   5 SQLCA_WARN0          CHAR(1),
   5 SQLCA_WARN1          CHAR(1),
   5 SQLCA_WARN2          CHAR(1),
   5 SQLCA_WARN3          CHAR(1),
   5 SQLCA_WARN4          CHAR(1),
   5 SQLCA_WARN5          CHAR(1),
   5 SQLCA_WARN6          CHAR(1),
   5 SQLCA_WARN7          CHAR(1);

```

**Note:** All REDEFINES are for compatibility with other implementations of SQL.

```

DCL 1 SQLCA,
   5 SQLCA_EYE_CATCH      CHAR(8) INIT('SQLCA***'),
   5 SQLCA_LEN_FIXED      BINARY(31) INIT(196),
   5 SQLCA_DB_VRS         CHAR(2) INIT('08'),
   5 SQLCA_DB_RLS         CHAR(2) INIT('10'),
   5 SQLCA_LUWID          CHAR(8) INIT(' '),
   5 SQLCODE              FIXED BINARY(31),
   5 SQLCA_ERR_LEN        FIXED BINARY(15),
   5 SQLCA_ERR_MSG        CHAR(80) INIT(' '),
   5 SQLCA_ERROR_PGM      CHAR(8) INIT(' '),
   5 SQLCA_FILLER_1       CHAR(2) INIT(' '),
   5 SQLCA_DSFCODE         CHAR(4) INIT(' '),
   5 SQLCA_INFICODE        FIXED BINARY(31),
   5 SQLCA_DBCODE_EXT      CHAR(2) INIT(' '),
   5 SQLCA_DBCODE_INT      FIXED BINARY(15),
   5 SQLCA_MISC_CODE1     FIXED BINARY(31),
   5 SQLCA_MISC_DATA       CHAR(8),
   5 SQLCA_WRN_AREA,
   10 SQLCA_WARNING (0:7) CHAR(1) INIT(' '),
   5 SQLCA_PGM_NAME        CHAR(8) INIT(' '),
   5 SQLCA_AUTHID          CHAR(18) INIT('authid here '),
   5 SQLCA_PLAN_NAME       CHAR(18) INIT('plan name here ');

```

```

DCL SQLCAID          CHAR(8)          DEF SQLCA_EYE_CATCH;
DCL SQLCABC          FIXED BINARY(31) DEF SQLCA_LEN;
DCL SQLERRML         FIXED BINARY(15) DEF SQLCA_ERR_LEN;
DCL SQLERRMC         CHAR(80)         DEF SQLCA_ERR_MSG;
DCL SQLERRP          CHAR(8)          DEF SQLCA_ERROR_PGM;
DCL SQLCA_SQLSTATE   CHAR(5)          DEF SQLCA_MISC_DATA,
SQLSTATE             CHAR(5)          DEF SQLCA_MISC_DATA;
DCL 1 SQLWARN        DEFINED SQLCA_WRN_AREA,
    5 SQLWARN0        CHAR(1),
    5 SQLWARN1        CHAR(1),
    5 SQLWARN2        CHAR(1),
    5 SQLWARN3        CHAR(1),
    5 SQLWARN4        CHAR(1),
    5 SQLWARN5        CHAR(1),
    5 SQLWARN6        CHAR(1),
    5 SQLWARN7        CHAR(1);
DCL 1 SQLCA_WARN     DEFINED SQLCA_WRN_AREA,
    5 SQLCA_WARN0     CHAR(1),
    5 SQLCA_WARN1     CHAR(1),
    5 SQLCA_WARN2     CHAR(1),
    5 SQLCA_WARN3     CHAR(1),
    5 SQLCA_WARN4     CHAR(1),
    5 SQLCA_WARN5     CHAR(1),
    5 SQLCA_WARN6     CHAR(1),
    5 SQLCA_WARN7     CHAR(1);

```

**Note:** All REDEFINES are for compatibility with other implementations of SQL.

### SQLCA - CA Datacom/DB Format (Assembler)

```

SQLCA  DSECT
SQLCAEYE DS  CL8  .EYE CATCHER
SQLCALEN DS  F    .BLOCK LENGTH
SQLCABV  DS  CL2  .DB VERSION
SQLCABR  DS  CL2  .DB RELEASE
SQLCALUW DS  CL8  .LUW ID
SQLCODE  DS  F    .SQL RETURN CODE
SQLCAERI DS  0CL82 .ERROR TEXT
SQLCAELN DS  H    . LENGTH
SQLCAEMS DS  CL80 . MESSAGE
SQLCAEPG DS  CL8  . PROGRAM
SQLCAFL1 DS  CL2  .UNUSED
SQLCAEDT DS  0CL22 .ERROR DATA
SQLCADSF DS  CL4  . DSF EXTERNAL CODE
SQLCAINF DS  F    . RESERVED
SQLCADBC DS  0CL6 . DB CODES

```



```

SQLCDBX DS    CL2    .    EXTERNAL
SQLCDBI DS    H      .    INTERNAL
SQLCAMC1 DS   F      .ROWS AFFECTED
SQLSTATE DS   0CL5  .SQLSTATE
SQLCAMC2 DS   F      .RESERVED
SQLCAMC3 DS   F      .RESERVED
SQLCAWRN DS   0CL8  .WARNINGS
SQLCAWN0 DS   CL1   .    SQLCA WARNING
SQLCAWN1 DS   CL1   .    RESERVED
SQLCAWN2 DS   CL1   .    RESERVED
SQLCAWN3 DS   CL1   .    UNEQUAL VARS
SQLCAWN4 DS   CL1   .    RESERVED
SQLCAWN5 DS   CL1   .    DATE/TIMESTAMP ADJUSTMENT
SQLCAWN6 DS   CL1   .    RESERVED
SQLCAWN7 DS   CL1   .    RESERVED
SQLCAPGM DS   CL8   .UNUSED
SQLCAATH DS   CL18  .AUTH ID
SQLCAPLN DS   CL18  .PLAN NAME
SQLCADLN EQU  *-SQLCA

```

**Note:** All REDEFINES are for compatibility with other implementations of SQL.

## SQLCA - (C Language)

Here is an example of the SQLCA in the C language.

```

struct sqlca {
    char    sqlca_eye_catch [8];
    #define sqlcaid    sqlca_eye_catch
    int     sqlca_len;
    #define sqlcabc    sqlca_len

    #ifndef DB2
        char    sqlca_db_vrs    [2];
        char    sqlca_db_rls    [2];
        char    sqlca_lwid     [8];
    #endif

    int     sqlca_code;
    #define sqlcode    sqlca_code
    #define sqlcade    sqlca_code

    #ifndef DB2
        short   sqlca_err_len;
        #define sqlerrml    sqlca_err_len
    #endif

    char    sqlca_err_msg SQLCA_MSG_LEN&hyphen. ;

```

```
#define sqlerrmc sqlca_err_msg
#define sqlerrm sqlca_err_msg
        char sqlca_error_pgm [8];
#define sqlerrp sqlca_error_pgm

#ifdef DB2
        int sqlerrd 6&hyphen. ;
#else
        char sqlca_filler_1 [2];
        char sqlca_dsfcodes [4];
        int sqlca_infcode;
        char sqlca_dbcode_ext [2];
        short sqlca_dbcode_int;
        int sqlca_misc_code1;
        char sqlca_sqlstate [5];
#define sqlstate sqlca_sqlstate
        char sqlca_filler_2 [3];
#endif
        char sqlca_wrn_area SQLCA_WARN_LEN&hyphen. ;
#define sqlca_warn0 sqlca_wrn_area[0]
#define sqlca_warn1 sqlca_wrn_area[1]
#define sqlca_warn2 sqlca_wrn_area[2]
#define sqlca_warn3 sqlca_wrn_area[3]
#define sqlca_warn4 sqlca_wrn_area[4]
#define sqlca_warn5 sqlca_wrn_area[5]
#define sqlca_warn6 sqlca_wrn_area[6]
#define sqlca_warn7 sqlca_wrn_area[7]
#define sqlwarn0 sqlca_wrn_area[0]
#define sqlwarn1 sqlca_wrn_area[1]
#define sqlwarn2 sqlca_wrn_area[2]
#define sqlwarn3 sqlca_wrn_area[3]
#define sqlwarn4 sqlca_wrn_area[4]
#define sqlwarn5 sqlca_wrn_area[5]
#define sqlwarn6 sqlca_wrn_area[6]
#define sqlwarn7 sqlca_wrn_area[7]
```

```

        #ifdef DB2
                char    sqlcext    [5];
        }
sqlca = {"SQLCA  ",136,0,"
        ",0,0,0,0,0,0,"
        ",0,0,0,0,0
};
        #else
                char    sqlca_pgm_name  [8];
                char    sqlca_authid   [18];
                char    sqlca_plan_name [18];
        }
sqlca = {"SQLCA***",196,"10", "0 ", "
        ", "
        ", " ", "
        ",0," ", 0,
        ', ', ', ', ', ', "
        ", "
        ", "CA AuthID", "<pgmname>"
};
        #endif

```

## Description of SQLCA in CA Datacom/DB Format

### Description of SQLCA in CA Datacom/DB Format:

Languages and Field Names	Descriptions
<b>COBOL:</b> SQLCA-EYE-CATCH <b>PL/I:</b> SQLCA_EYE_CATCH <b>Assembler:</b> SQLCAEYE <b>C:</b> sqlca_eye_catch	A core mark to help find the SQLCA in diagnostic situations (containing 'SQLCA***' in COBOL).
<b>COBOL:</b> SQLCA-LEN <b>PL/I:</b> SQLCA_LEN <b>Assembler:</b> SQLCALEN <b>C:</b> sqlca_len	The length of the SQLCA (196 in COBOL).
<b>COBOL:</b> SQLCA-DB-VRS <b>PL/I:</b> SQLCA_DB_VRS <b>Assembler:</b> SQLCADBV <b>C:</b> sqlca_db_vrs	The CA Datacom/DB version.
<b>COBOL:</b> SQLCA-DB-RLS <b>PL/I:</b> SQLCA_DB_RLS <b>Assembler:</b> SQLCADBR <b>C:</b> sqlca_db_rls	The CA Datacom/DB release.
<b>COBOL:</b> SQLCA-LUWID <b>PL/I:</b> SQLCA_LUWID <b>Assembler:</b> SQLCALUW <b>C:</b> sqlca_luwid	Reserved.

Languages and Field Names	Descriptions
<b>COBOL:</b> SQLCODE <b>PL/I:</b> SQLCA_CODE (ANSI) <b>PL/I:</b> SQLCODE (non-ANSI) <b>Assembler:</b> SQLCODE <b>C:</b> sqlca_code or sqlcode or sqlcade	Value returned from an SQL call.  If an exception declaration (WHENEVER statement) is not provided, the recommended practice is that your program include code to check the returned value immediately after each executable SQL statement.
<b>COBOL:</b> SQLCA-ERROR-INFO <b>PL/I:</b> SQLCA_ERROR_INFO <b>Assembler:</b> SQLCAERI <b>C:</b> N/A (See following error fields for C.)	The error information area.
<b>COBOL:</b> SQLCA-ERROR-LEN <b>PL/I:</b> SQLCA_ERROR_LEN <b>Assembler:</b> SQLCAELN <b>C:</b> sqlca_err_len	The length of the error return string.
<b>COBOL:</b> SQLCA-ERROR-MSG <b>PL/I:</b> SQLCA_ERROR_MSG <b>Assembler:</b> SQLCAEMS <b>C:</b> sqlca_err_msg	A brief description of the error.
<b>COBOL:</b> SQLCA-ERROR-PGM <b>PL/I:</b> SQLCA_ERROR_PGM <b>Assembler:</b> SQLCAEPG <b>C:</b> sqlca_err_pgm	Contains the name of the CA Datacom/DB module which reported the error.
<b>COBOL:</b> SQLCA-FILLER-1 <b>PL/I:</b> SQLCA_FILLER_1 <b>Assembler:</b> SQLCAFL1 <b>C:</b> sqlca_filler_1	In PL/I and Assembler, this is a place holder.
<b>COBOL:</b> SQLCA-ERROR-DATA <b>PL/I:</b> SQLCA_ERROR_DATA <b>Assembler:</b> SQLCAEDT <b>C:</b> N/A (See following code fields for C.)	Provides diagnostic information.
<b>COBOL:</b> SQLCA-DSFCODE <b>PL/I:</b> SQLCA_DSFCODE <b>Assembler:</b> SQLCADSF <b>C:</b> sqlca_dsfcode	The return code from the CA Datacom Datadictionary Service Facility.
<b>COBOL:</b> SQLCA-INFICODE <b>PL/I:</b> SQLCA_INFICODE <b>Assembler:</b> SQLCAINF <b>C:</b> sqlca_infcode	Reserved for future use.
<b>COBOL:</b> SQLCA-DBCOD <b>PL/I:</b> SQLCA_DBCOD <b>Assembler:</b> SQLCADBC <b>C:</b> sqlca_dbcode	Contains additional return codes from CA Datacom/DB for some error conditions, to aid in diagnosing errors.

Languages and Field Names	Descriptions
<b>COBOL:</b> SQLCA-DBCOD-EXT <b>PL/I:</b> SQLCA_DBCOD-EXT <b>Assembler:</b> SQLCDBEX <b>C:</b> sqlca_dbcode_ext	The CA Datacom/DB external return code.
<b>COBOL:</b> SQLCA-DBCOD-INT <b>PL/I:</b> SQLCA_DBCOD-INT <b>Assembler:</b> SQLCDBI <b>C:</b> sqlca_dbcode_int	The CA Datacom/DB internal return code.
<b>COBOL:</b> SQLCA-MISC-DATA <b>PL/I:</b> SQLCA_MISC-DATA <b>Assembler:</b> SQLCAMC1 <b>C:</b> sqlca_misc_code1	Number of rows affected by an UPDATE, INSERT or DELETE statement.
<b>COBOL:</b> SQLCA-MISC-CODE2 <b>COBOL:</b> SQLCA-MISC-CODE3 <b>PL/I:</b> SQLCA_MISC-CODE2 <b>PL/I:</b> SQLCA_MISC-CODE3	redefined (following) to SQLSTATE field.
<b>COBOL:</b> SQLSTATE <b>PL/I:</b> SQLSTATE <b>Assembler:</b> SQLSTATE <b>C:</b> sqlca_sqlstate or sqlstate	The field in which the ANSI-compatible SQLSTATE return code is supplied.
<b>Assembler:</b> SQLCAMC2 <b>Assembler:</b> SQLCAMC3 <b>C:</b> sqlca_filler_2 <b>C:</b> sqlca_misc_code2	Reserved.
<b>COBOL:</b> SQLCA-WARNING <b>PL/I:</b> SQLCA_WARNING <b>Assembler:</b> SQLCAWRN <b>C:</b> sqlca_wrn_area	An array of eight characters which provides warning return codes (a W indicates that a warning has been returned). For tables that explain the significance of the return code for each element in the array, see: <ul style="list-style-type: none"> <li>■ <a href="#">Warning Array - COBOL</a> (see page 254) (for COBOL)</li> <li>■ <a href="#">Warning Arrays - PL/I, Assembler, C</a> (see page 255) (for PL/I, Assembler, and C).</li> </ul>
<b>COBOL:</b> SQLCA-PGM-NAME <b>PL/I:</b> SQLCA_PGM-NAME <b>Assembler:</b> SQLCAPGM <b>C:</b> sqlca_pgm_name	Contains the program identification assigned in the PROGRAM-ID statement.
<b>COBOL:</b> SQLCA-AUTHID <b>PL/I:</b> SQLCA_AUTHID <b>Assembler:</b> SQLCAATH <b>C:</b> sqlca_authid	Contains the authorization ID specified in the Preprocessor AUTHID= option.

Languages and Field Names	Descriptions
<b>COBOL:</b> SQLCA-PLAN-NAME <b>PL/I:</b> SQLCA_PLAN_NAME <b>Assembler:</b> SQLCAPLN <b>C:</b> sqlca_plan_name	Contains the name of the plan specified in the Preprocessor PLANAME= (or PLANNAME= in PL/I) option. (In COBOL, if the option is not specified, the default is the PROGRAM-ID.)
<b>Assembler:</b> SQLCADLN	The length of the SQLCA.

## Warning Array - COBOL

### Warning Return Code Array—COBOL:

Array Element	Explanation
SQLCA-WARNING(1)	After executing a DBSQLE command this field contains a W if any warning condition has been detected. Check the other warning flags for the specific warning. Otherwise contains spaces, that is, x'40'.
SQLCA-WARNING(2)	Contains W if the value of a string column was truncated when assigned to a host-variable. Otherwise contains spaces, that is, x'40'.
SQLCA-WARNING(3)	Contains W if null values were eliminated from the argument of a column function; not necessarily set to W for the MIN function because its results are not dependent on the elimination of null values. Otherwise contains spaces, that is, x'40'.
SQLCA-WARNING(4)	Contains a W if the number of host-variables in the INTO clause of a FETCH or SELECT INTO statement is not equal to the number of items in the SELECT list. The actual number of columns returned is the smaller of the two numbers. Otherwise contains spaces, that is, x'40'.
SQLCA-WARNING(5)	Contains W if a prepared UPDATE or DELETE statement does not include a WHERE clause. Otherwise contains spaces, that is, x'40'.
SQLCA-WARNING(6)	Reserved for future use.
SQLCA-WARNING(7)	Contains a W if an adjustment was made to a DATE or TIMESTAMP value to correct an invalid date resulting from an arithmetic operation. Otherwise contains spaces, that is, x'40'.
SQLCA-WARNING(8)	Contains a W if it was generated by a procedure.

**Note:** In CA Datacom/DB Release 8.0, SQLCA-WARNING(6) was the DATE/TIME adjustment warning, but with later versions that warning is in SQLCA-WARNING(7). Application programs written for use with Release 8.0 may therefore have to be adjusted accordingly for use with later versions.

## Warning Arrays - PL/I, Assembler, C

### Warning Return Code Array— PL/I, Assembler, C:

Language and Array Element	Explanation
<b>PL/I:</b> SQLCA_WARNING0 <b>Assembler:</b> SQLCAWN0 <b>C:</b> sqlwarn0	Set to W if any of SQLCA-WARNING (2)—(8) is set to W.
<b>PL/I:</b> SQLCA_WARNING1 <b>Assembler:</b> SQLCAWN1 <b>C:</b> sqlwarn1	Reserved for future use.
<b>PL/I:</b> SQLCA_WARNING2 <b>Assembler:</b> SQLCAWN2 <b>C:</b> sqlwarn2	Reserved for future use.
<b>PL/I:</b> SQLCA_WARNING3 <b>Assembler:</b> SQLCAWN3 <b>C:</b> sqlwarn3	Set to W if the number of items in the SELECT list is not equal to the number of variables in the INTO clause. The number of items returned is the lesser of these two numbers.
<b>PL/I:</b> SQLCA_WARNING4 <b>Assembler:</b> SQLCAWN4 <b>C:</b> sqlwarn4	Reserved for future use.
<b>PL/I:</b> SQLCA_WARNING5 <b>Assembler:</b> SQLCAWN5 <b>C:</b> sqlwarn5	Set to W if an adjustment was made to a DATE or TIMESTAMP value to correct an invalid date resulting from an arithmetic operation.
<b>PL/I:</b> SQLCA_WARNING6 <b>Assembler:</b> SQLCAWN6 <b>C:</b> sqlwarn6	Reserved for future use.
<b>PL/I:</b> SQLCA_WARNING7 <b>Assembler:</b> SQLCAWN7 <b>C:</b> sqlwarn7	Contains a W if it was generated by a procedure.

## SQLCA - DB2 Format (COBOL)

```

01  SQLCA.
    05  SQLCAID                PIC X(08).
    05  SQLCABC                PIC S9(9) COMP VALUE +136.
    05  SQLCODE                PIC S9(9) COMP VALUE +0.
    05  SQLERRM.
        49  SQLERRML          PIC S9(4) COMP.
        49  SQLERRMC          PIC X(70).
    05  SQLERRP                PIC X(08).
    05  SQLERRD                PIC S9(9) COMP OCCURS 6 TIMES.
    05  SQLWARN.
        10  SQLWARN0          PIC X.
        10  SQLWARN1          PIC X.
        10  SQLWARN2          PIC X.
        10  SQLWARN3          PIC X.
        10  SQLWARN4          PIC X.
        10  SQLWARN5          PIC X.
        10  SQLWARN6          PIC X.
        10  SQLWARN7          PIC X.
    05  SQLEXT.
        10  SQLWARN8          PIC X(1).
        10  SQLWARN9          PIC X(1).
        10  SQLWARNA          PIC X(1).
        10  SQLSTATE          PIC X(5).

```

## SQLCA - DB2 Format (PL/I)

```

DCL 1 SQLCA,
    2 SQLCAID                CHAR(8) INIT('SQLCA'),
    2 SQLCABC                BIN FIXED(31) INIT(136),
    2 SQLCADE                BIN FIXED(31) INIT(0),
    2 SQLERRM                CHAR(70) VAR,
    2 SQLERRP                CHAR(8),
    2 SQLERRD(6)            BIN FIXED(31) INIT(0),
    2 SQLCA_WRN_AREA,
        3 SQLCA_WARNING      (0:10) CHAR(1) INIT(' '),
    2 SQLEXT                CHAR(5);
DCL SQLCA_EYE_CATCH        CHAR(8)
    DEFINED SQLCAID;
DCL SQLCA_LEN FIXED        BIN(31)
    DEFINED SQLCABC;
DCL SQLCA_ERROR_PGM        CHAR(8)
    DEFINED SQLERRP;
DCL 1 SQLWARN              DEFINED SQLCA_WRN_AREA,

```



```

5 SQLWARN0      CHAR(1),
5 SQLWARN1      CHAR(1),
5 SQLWARN2      CHAR(1),
5 SQLWARN3      CHAR(1),
5 SQLWARN4      CHAR(1),
5 SQLWARN5      CHAR(1),
5 SQLWARN6      CHAR(1),
5 SQLWARN7      CHAR(1),
5 SQLWARN8      CHAR(1),
5 SQLWARN9      CHAR(1),
5 SQLWARNA      CHAR(1);
DCL 1 SQLCA_WARN DEFINED SQLCA_WRN_AREA,
5 SQLCA_WARN0    CHAR(1),
5 SQLCA_WARN1    CHAR(1),
5 SQLCA_WARN2    CHAR(1),
5 SQLCA_WARN3    CHAR(1),
5 SQLCA_WARN4    CHAR(1),
5 SQLCA_WARN5    CHAR(1),
5 SQLCA_WARN6    CHAR(1),
5 SQLCA_WARN7    CHAR(1),
5 SQLCA_WARN8    CHAR(1),
5 SQLCA_WARN9    CHAR(1),
5 SQLCA_WARNA    CHAR(1);

```

### DB2 format (non-ANSI)

```

DCL 1 SQLCA,
2 SQLCAID      CHAR(8) INIT('SQLCA'),
2 SQLCABC      BIN FIXED(31) INIT(136),
2 SQLCODE      BIN FIXED(31) INIT(0),
2 SQLERRM      CHAR(70) VAR,
2 SQLERRP      CHAR(8),
2 SQLERRD(6)   BIN FIXED(31) INIT(0),
2 SQLCA_WRN_AREA,
3 SQLCA_WARNING (0:10) CHAR(1) INIT(' '),
2 SQLEXT      CHAR(5);
DCL SQLCA_EYE_CATCH CHAR(8)
      DEFINED SQLCAID;
DCL SQLCA_LEN FIXED BIN(31)
      DEFINED SQLCABC;
DCL SQLCA_ERROR_PGM CHAR(8)
      DEFINED SQLERRP;
DCL 1 SQLWARN   DEFINED SQLCA_WRN_AREA,

```

```

5 SQLWARN0      CHAR(1),
5 SQLWARN1      CHAR(1),
5 SQLWARN2      CHAR(1),
5 SQLWARN3      CHAR(1),
5 SQLWARN4      CHAR(1),
5 SQLWARN5      CHAR(1),
5 SQLWARN6      CHAR(1),
5 SQLWARN7      CHAR(1),
5 SQLWARN8      CHAR(1),
5 SQLWARN9      CHAR(1),
5 SQLWARNA      CHAR(1);
DCL 1 SQLCA_WARN DEFINED SQLCA_WRN_AREA,
5 SQLCA_WARN0    CHAR(1),
5 SQLCA_WARN1    CHAR(1),
5 SQLCA_WARN2    CHAR(1),
5 SQLCA_WARN3    CHAR(1),
5 SQLCA_WARN4    CHAR(1),
5 SQLCA_WARN5    CHAR(1),
5 SQLCA_WARN6    CHAR(1),
5 SQLCA_WARN7    CHAR(1),
5 SQLCA_WARN8    CHAR(1),
5 SQLCA_WARN9    CHAR(1),
5 SQLCA_WARNA    CHAR(1);

```

### SQLCA - DB2 Format (Assembler)

```

SQLCA  DSECT
SQLCAID DS  CL8  .EYE CATCHER
SQLCABC DS  F    .BLOCK LENGTH
SQLCODE DS  F    .SQL RETURN CODE
SQLERRM DS  H,CL70 .ERROR
SQLERRP DS  CL8  .ERROR PROGRAM
SQLERRD DS  6F   .ERROR CODES
SQLWARN DS  0C   .WARNINGS
SQLWARN0 DS  CL1 . WARNING
SQLWARN1 DS  CL1 . WARNING
SQLWARN2 DS  CL1 . WARNING
SQLWARN3 DS  CL1 . WARNING
SQLWARN4 DS  CL1 . WARNING
SQLWARN5 DS  CL1 . WARNING
SQLWARN6 DS  CL1 . WARNING
SQLWARN7 DS  CL1 . WARNING
SQLEXT  DS  CL8
      ORG SQLEXT
SQLWARN8 DS  CL1 . WARNING
SQLWARN9 DS  CL1 . WARNING
SQLWARNA DS  CL1 . WARNING
SQLSTATE DS  CL5 .
SQLCA2LN EQU  *-SQLCA

```

## SQLCA - (C Language)

See [SQLCA - \(C Language\)](#) (see page 249).

## Description of SQLCA in DB2 Format

**Description of SQLCA in DB2 Format:**

<b>COBOL</b>	<b>PL/I</b>	<b>Assembler</b>	<b>Description</b>
SQLCAID	SQLCAID	SQLCAID	An eye-catcher for storage dumps, set to SQLCA.
SQLCABC	SQLCABC	SQLCABC	Must be set to the length of the SQLCA: 136.
SQLCODE	SQLCADE (ANSI) SQLCODE (non-ANSI)	SQLCODE	Value returned from an SQL call. If an exception declaration (WHENEVER statement) is not provided, the recommended practice is that your program include code to check the SQLCODE value immediately after each executable SQL statement.
SQLERRML	SQLERRM	SQLERRM	In COBOL, contains the length indicator for SQLERRMC in the range of zero through 70. It contains zero if there is no error or exception condition, in which case the value of SQLERRMC is not pertinent.  In PL/I, contains the length of the error return string and a brief description of the error.  In Assembler, contains the length indicator in the range of zero through 70. It contains zero if there is no error or exception condition. If greater than zero, up to 70 bytes of error text follows.
SQLERRMC			In COBOL, contains one or more tokens, separated by X'FF', that are substituted for variables in the descriptions of error conditions.
SQLERRP	SQLERRP	SQLERRP	Contains the name of the CA Datacom/DB module which reported the error.
		SQLERRD	In Assembler, this is the top of error codes (not an actual field).
SQLERRD(1)	SQLERRD(1)	SQLERRD1	Contains the CA Datacom/DB SQLCODE to aid in diagnosing errors. The value in the SQLCODE field is the DB2 equivalent of this value.
SQLERRD(2)	SQLERRD(2)	SQLERRD2	Contains the CA Datacom/DB external return code for some error conditions, to aid in diagnosing errors.

<b>COBOL</b>	<b>PL/I</b>	<b>Assembler</b>	<b>Description</b>
SQLERRD(3)	SQLERRD(3)	SQLERRD3	Contains the number of rows affected after an INSERT, UPDATE, or DELETE (but not rows deleted as a result of CASCADE delete).
SQLERRD(4)	SQLERRD(4)	SQLERRD4	Reserved for future use.
SQLERRD(5)	SQLERRD(5)	SQLERRD5	Reserved for future use.
SQLERRD(6)	SQLERRD(6)	SQLERRD6	Contains the CA Datacom/DB internal return code for some error conditions, to aid in diagnosing errors.
		SQLWARN	In Assembler, this is the top of warnings (not an actual field).
SQLWARN0	SQLWARN0	SQLWARN0	After executing a DBSQL command this field contains a W if any warning condition has been detected. Check the other warning flags for the specific warning. Otherwise contains spaces, that is, x'40'.
SQLWARN1	SQLWARN1	SQLWARN1	Contains W if the value of a string column was truncated when assigned to a host-variable. Otherwise contains spaces, that is, x'40'.
SQLWARN2	SQLWARN2	SQLWARN2	Contains W if null values were eliminated from the argument of a column function; not necessarily set to W for the MIN function because its results are not dependent on the elimination of null values. Otherwise contains spaces, that is, x'40'.
SQLWARN3	SQLWARN3	SQLWARN3	Contains a W if the number of host-variables in the INTO clause of a FETCH or SELECT INTO statement is not equal to the number of items in the SELECT list. The actual number of columns returned is the smaller of the two numbers. Otherwise contains spaces, that is, x'40'.
SQLWARN4	SQLWARN4	SQLWARN4	Contains W if a prepared UPDATE or DELETE statement does not include a WHERE clause. Otherwise contains spaces, that is, x'40'.
SQLWARN5	SQLWARN5	SQLWARN5	Reserved for future use.
SQLWARN6	SQLWARN6	SQLWARN6	Contains a W if an adjustment was made to a DATE or TIMESTAMP value to correct an invalid date resulting from an arithmetic operation. Otherwise contains spaces, that is, x'40'.
SQLWARN7	SQLWARN7	SQLWARN7	Contains a W if it was generated by a procedure.
SQLEXT	SQLEXT	SQLEXT	Reserved for future use.

COBOL	PL/I	Assembler	Description
SQLWARN8	SQLWARN8	SQLWARN8	Reserved for future use.
SQLWARN9	SQLWARN9	SQLWARN9	Reserved for future use.
SQLWARNA	SQLWARNA	SQLWARNA	Reserved for future use.
SQLSTATE	SQLSTATE	SQLSTATE	In COBOL and PL/I, this is the ANSI format return code. In Assembler, this is the execution error return code.
		SQLCA2LN	In Assembler, the length of the SQLCA.

## SQL Work Area (SQLWA)

The SQL Work Area (SQLWA) is a collection of variables used by CA Datacom/DB to provide an application program with information about each of its SQL statements. The CA Datacom/DB Preprocessor generates one SQLWA for each compiled embedded SQL statement.

**Important!** Do not change any of the Preprocessor generated output pertaining to the SQLWAs.

## SQLWA Examples

SQLWA examples are provided in the following pages.

### SQLWA in COBOL

See [SQLWA - COBOL](#) (see page 262) for an example COBOL version of the SQLWA.

### SQLWA in PL/I

With the CA Datacom/DB SQL Preprocessor for PL/I there are two possible formats of the SQLWA:

- CA Datacom/DB format (see [SQLWA - db. Format \(PL/I\)](#) (see page 263))
- DB2 format (see [SQLWA - DB2 Format \(PL/I\)](#) (see page 264))

## SQLWA in Assembler

With the CA Datacom/DB SQL Preprocessor for Assembler there are two possible formats of the SQLWA:

- CA Datacom/DB format (see [SQLWA - CA Datacom/DB Format \(Assembler\)](#) (see page 265))
- DB2 format (see [SQLWA - DB2 Format \(Assembler\)](#) (see page 266))

## SQLWA in C

For an example of the SQLWA format in the C language, see [SQLWA - Format for C Language](#) (see page 267).

## SQLWA - COBOL

Following is the COBOL version of an SQLWA. In the example, *n* represents the number for each SQLWA generated by the Preprocessor, while *x* represents the number generated for each host-variable referenced in the SQL statement.

```
01  SQLWAn.  
   05  SQLWAn-EYE-CATCH          PIC X(08) VALUE 'SQLWA***'.  
   05  SQLWAn-LEN                PIC S9(9) COMP VALUE +48.  
   05  SQLWAn-COMMAND           PIC X(05) VALUE 'QEXEC'.  
   05  SQLWAn-FILLER            PIC X(03).  
   05  SQLWAn-PROC-NAME         PIC X(08) VALUE SPACES.  
   05  SQLWAn-CURS-T            PIC S9(9) COMP VALUE +0.  
   05  SQLWAn-STMT-ID  
       PIC S9(9) COMP VALUE +0.  
   05  SQLWAn-ADDR-HOST-DESC     PIC S9(9) COMP VALUE +0.  
   05  SQLWAn-END-CATCH         PIC X(08) VALUE 'ENDSQLWA'.  
   05  HOST-VARn-AREA.  
       10  HOST-VARn-LEN        PIC S9(9) COMP VALUE +104.  
       10  HOST-VARn-NBR-ENT    PIC S9(4) COMP VALUE +6.  
       10  HOST-VARn-FILLER     PIC X(02).
```

```

10 HOST-VARn-DESC.
15 HOST-VARn-x-TYPE
PIC S9(4) COMP VALUE +452.
15 HOST-VARn-x-LEN
PIC S9(4) COMP VALUE +3.
15 HOST-VARn-x-DATA
PIC S9(9) COMP.
15 HOST-VARn-x-IND
PIC S9(9) COMP.
15 HOST-VARn-x-DIR
PIC X(01) VALUE 'F'.
88 HOST-VARn-x-TO-MUF
VALUE 'T'.
88 HOST-VARn-x-FROM-MUF
VALUE 'F'.
15 HOST-VARn-x-FILLER
PIC X(03).

```

## SQLWA - CA Datacom/DB Format (PL/I)

Following is the CA Datacom/DB format of the PL/I version of an SQLWA. In the example, *n* represents the number for each SQLWA generated by the Preprocessor.

```

DCL 1 SQLWAn,
5 SQLWAn_EYE_CATCH          CHAR(8) INIT('SQLWA***'),
5 SQLWAn_LEN                FIXED BINARY(31) INIT(48),
5 SQLWAn_COMMAND            CHAR(5) INIT('QEXEC'),
5 SQLWAn_FLAGS              BIT(8) INIT(0),
5 SQLWAn_FILLER             CHAR(2) INIT(' '),
5 SQLWAn_PROC_NAME          CHAR(8) INIT(' '),
5 SQLWAn_STAMP              FIXED BINARY(31) INIT(nnnnnnnn),
5 SQLWAn_STMT_ID            FIXED BINARY(31) INIT(nnnn),
5 SQLWAn_ADDR_HOST_DESC     POINTER,
5 SQLWAn_END_CATCH          CHAR(8) INIT('ENDSQLWA'),
5 HOST_VARn_AREA,
10 HOST_VARn_LEN            FIXED BINARY(31) INIT(nnn),
10 HOST_VARn_NBR_ENT        FIXED BINARY(15) INIT(nn),
10 HOST_VARn_FILLER         CHAR(2) INIT(' '),
10 HOST_VARn_DESC,
/* host field name m */
15 HOST_VARn_m_TYPE         FIXED BINARY(15) INIT(%t),
15 HOST_VARn_m_LEN          FIXED BINARY(15) INIT(%u),
15 HOST_VARn_m_DATA         POINTER,
15 HOST_VARn_m_IND          POINTER,
15 HOST_VARn_m_DIR          CHAR(1) INIT('d'),
15 HOST_VARn_m_FILR         CHAR(3) INIT(' '),

```

## SQLWA - DB2 Format (PL/I)

Following is the DB2 format of the PL/I version of an SQLWA. In the example, *n* represents the number for each SQLWA generated by the Preprocessor.

```

DCL 1 SQLWAn,
    5 SQLWAn_EYE_CATCH          CHAR(8) INIT('SQLWA***'),
    5 SQLWAn_LEN                FIXED BINARY(31) INIT(96),
    5 SQLWAn_COMMAND           CHAR(5) INIT('QEXEC'),
    5 SQLWAn_FLAGS              BIT(8) INIT(1),
    5 SQLWAn_FILLER            CHAR(2) INIT(' '),
    5 SQLWAn_PROC_NAME          CHAR(8) INIT(' '),
    5 SQLWAn_STAMP              FIXED BINARY(31) INIT(nnnnnnn),
    5 SQLWAn_STMT_ID            FIXED BINARY(31) INIT(nnnn),
    5 SQLWAn_ADDR_HOST_DESC     POINTER,
    5 SQLWAn_PGM_NAME           CHAR(8) INIT(' '),
    5 SQLWAn_AUTHID             CHAR(18) INIT('authid here '),
    5 SQLWAn_PLAN_NAME          CHAR(18) INIT('plan name here '),
    5 SQLWAn_DB_VRS             CHAR(2) INIT(%d),
    5 SQLWAn_DB_RLS             CHAR(2) INIT(%e),
    5 SQLWAn_END_CATCH          CHAR(8) INIT('ENDSQLWA'),
    5 HOST_VARn_AREA,
        10 HOST_VARn_LEN        FIXED BINARY(31) INIT(nnn),
        10 HOST_VARn_NBR_ENT    FIXED BINARY(15) INIT(nn),
        10 HOST_VARn_FILLER     CHAR(2) INIT(' '),
        10 HOST_VARn_DESC,
            /* host field name m */
            15 HOST_VARn_m_TYPE  FIXED BINARY(15) INIT(%t),
            15 HOST_VARn_m_LEN   FIXED BINARY(15) INIT(%u),
            15 HOST_VARn_m_DATA  POINTER,
            15 HOST_VARn_m_IND   POINTER,
            15 HOST_VARn_m_DIR   CHAR(1) INIT('d'),
            15 HOST_VARn_m_FILR  CHAR(3) INIT(' '),

```



## SQLWA - CA Datacom/DB Format (Assembler)

In the following example, n represents the number for each SQLWA generated by the Preprocessor for Assembler.

SQLWA	DSECT		
SQLWAEYE DS	CL8		.EYE CATCHER
SQLWALEN DS	F		.BLOCK LENGTH
SQLWACMD DS	CL5		.Q COMMAND
SQLWAFLG DS	X		.FLAGS
SQLWAFIL DS	CL2		.UNUSED
SQLWAFNM DS	CL8		.ROUTINE NAME
SQLWASTP DS	F		.STAMP
SQLWASID DS	F		.STATEMENT ID
SQLWAHDS DS	F		.ADDR HOST VAR AREA
SQLWAEND DS	CL8		.END EYE CATCHER
SQLWADLN EQU	*-SQLWA		
SQLHAARA DS	0CL8		.HOST VAR AREA
SQLHAHVL DS	F		.LENGTH OF VARS
SQLHANHV DS	H		.NBR OF VARS
SQLHAFIL DS	CL2		.UNUSED
SQLHAALN EQU	*-SQLHAARA		.LENGTH OF HOST VAR AREA
SQLHVBGN DS	0C		.START OF VARS
SQLWATLN EQU	*-SQLWA		.LENGTH OF SQLWA & HOST HDR

0 to n VARs depending  
on statement

SQLHVTYP DS	H		.STORAGE TYPE
SQLHVLEN DS	H		.LENGTH OR PRECISION/SCALE
SQLHVDAT DS	F		.ADDR OF RECEIVING FIELD
SQLHVIND DS	F		.ADDR OF INDICATOR FIELD
SQLHVDIR DS	CL1		.DIRECTION
SQLHVFIL DS	CL3		.UNUSED
SQLHVDLN EQU	*-SQLHVAR		.LENGTH OF HOST VAR

## SQLWA - DB2 Format (Assembler)

SQLWA	DSECT		
SQLWAEYE DS	CL8	.EYE CATCHER	
SQLWALEN DS	F	.BLOCK LENGTH	
SQLWACMD DS	CL5	.COMMAND	
SQLWAFLG DS	X	.FLAGS /* always x'01' */	
SQLWAFIL DS	CL2	.UNUSED	
SQLWARTN DS	CL8	.ROUTINE NAME	
SQLWASTP DS	F	.STAMP	
SQLWASID DS	F	.STATEMENT ID	
SQLWAHDS DS	F	.ADDR HOST VAR AREA	
SQLWAPGM DS	CL8	.UNUSED	
SQLWAATH DS	CL18	.AUTH ID	
SQLWAPLN DS	CL18	.PLANNAME	
SQLWADBV DS	CL2	.VERSION	
SQLWADVR DS	CL2	.RELEASE	
SQLWAEND DS	CL8	.END EYE CATCHER	
SQLWADLN EQU	*-SQLWA		
SQLHAARA DS	0CL8	.HOST VAR AREA	
SQLHAHVL DS	F	.LENGTH OF VARS	
SQLHANHV DS	H	.NBR OF VARS	
SQLHAFIL DS	CL2	.UNUSED	
SQLHAALN EQU	*-SQLHAARA	.LENGTH OF HOST VAR AREA	
SQLHVBN DS	0C	.START OF VARS	
SQLWATLN EQU	*-SQLWA	.LENGTH OF SQLWA & HOST HDR	

0 to n VARs depending  
on statement

SQLHVTYP DS	H	.STORAGE TYPE	
SQLHVLEN DS	H	.LENGTH OR PRECISION/SCALE	
SQLHVDAT DS	F	.ADDR OF RECEIVING FIELD	
SQLHVIND DS	F	.ADDR OF INDICATOR FIELD	
SQLHVDIR DS	CL1	.DIRECTION	
SQLHVFIL DS	CL3	.UNUSED	
SQLHVDLN EQU	*-SQLHVAR	.LENGTH OF HOST VAR	

## SQLWA - Format for C Language

Following is an example SQL Work Area (SQLWA) for the C language.

```

struct SQLwa
{
    char    eye_catch[8];
    int     len;
    char    command[5];
    unsigned char flags;
    char    filler[2];
    char    proc_name[8];
    int     stamp;
    int     stmt_id;
    char    *addr_host_desc;
    char    end_catch[8];
} sqlwa = {"SQLWA***", 48,"QEXEC", 0,
          " ", "", 70488250, 0, 0, "ENDSQLWA"};

struct
{
    int     len;
    short   nbr_ent;
    char    filler[2];
    struct sqlhvar
    {
        short   type;
        short   len;
        void    *data;
        short   *ind;
        char    dir;
        char    filr[3];
    } var[n];
} sqlHost;
char host_var_end[6] = "VAREND";

```

**Note:** The n in var[n] appears as a number that has a *special purpose*, that is, it corresponds to the maximum number of host variables needed by statements in the program.

## Error Handling

If the Preprocessor detects an SQL error that causes the program to be nonexecutable, the Preprocessor terminates processing. The plan being built and the statements are backed out. The previous plan (if one existed) for the program is restored. See the *CA Datacom/DB Message Reference Guide* for a list of possible return codes generated by the Preprocessors.

In COBOL, if the Preprocessor completes with a condition code other than 0, you should check SYSPRINT for the messages and the SQLCODE. The Preprocessor returns the following condition, depending on your environment:

### **z/OS**

Condition code 12

### **z/VSE**

Abnormal termination without a dump

For PL/I, C, and Assembler, all return codes that are equal to or greater than 16 indicate that the Preprocessor encountered an abnormal problem and terminated.

## Interaction of Multiple Preprocessors

Multiple preprocessors or precompilers may be used at your site. The other preprocessors or precompilers replace specialized code with legal statements. Source statements for the CA Datacom/DB SQL Preprocessor may be generated by another preprocessor, such as CA Librarian. Any preprocessor before the CA Datacom/DB SQL Preprocessor must be able to accept and pass through SQL statements.

Source code processed by the SQL Preprocessor could be processed by additional preprocessors after being preprocessed and before being compiled. There may be limitations on the forms of source statements that can be passed through the Preprocessor. For example, literals and comments not accepted by the supported host compilers might interfere with precompiler source scanning and cause errors. Interaction problems can only occur if one of the preprocessors or precompilers adds logic to the program which may be affected by another precompiler or preprocessor. You reduce the possibility of encountering interaction problems if you run the preprocessors or precompilers *in the following order*:

1. CA Librarian

CA Librarian retrieves the source code from a CA Librarian master file and passes it to the next preprocessor or precompiler. If your site has CA Librarian, you can extract the source code by running the batch CA Librarian program with the EXEC option. Use this method if the program contains COPYDD statements.

Alternatively, if the CA Librarian Access Method (LIB/AM) is installed and the program contains no COPYDD statements, you can omit the CA Librarian step and allow LIB/AM to pass the source statements directly to the preprocessor or precompile. For more information on using CA Librarian, see the appropriate CA Librarian documentation.

2. CA MetaCOBOL+ precompiler (*COBOL only*)

Because this precompiler modifies source code (non-SQL), it must be run before the CA Datacom/DB SQL Preprocessor for COBOL.

3. CA Datacom/DB SQL Preprocessors

One of these is run next because it replaces SQL statements with COBOL, PL/I, or Assembler statements.

4. CICS precompiler (*COBOL and PL/I only*)

Always run this precompiler last.

If one or more preprocessors or precompilers is not run, you must retain the order previously specified for the remaining preprocessors or precompilers. If your site has preprocessors or precompilers in addition to those previously listed, the order in which they run may vary.

## SQL Plan Options Special Topics

The following comments apply regardless of the host language used.

### Read-Only

If you choose the SQL Preprocessor option `ISOLEVEL=U`, the access plan is read-only and your application cannot execute the SQL statements `INSERT`, `UPDATE`, or `DELETE`.

In addition, a share lock is *not* acquired for rows accessed with `SELECT INTO` or through a cursor, which means you may access rows inserted or updated by other concurrent tasks that have not been committed and may therefore be backed out.

### Mixing Isolation Levels

You can not mix different isolation levels in the same unit of recovery. For example, if a program which uses isolation level C calls another program preprocessed with isolation level U, an SQL -144 return code (`INVALID TRANSACTION ISOLEVEL`) will be returned whenever a statement is executed from the second program unless the first program executes a `COMMIT WORK` statement to end the unit of recovery before calling the second program.

**Note:** The *both* parameter of the `SQLOPTION MUF` startup option controls whether you are allowed to mix the use of isolation level U and C plans under a single logical unit of work (LUW) for those LUWs not running any `SQLMODE ANSI` or `FIPS` plans. Specifying `YES` for the *both* option of `SQLOPTION` indicates that mixing is allowed. It is your responsibility to verify proper locking is performed and the database state is not updated incorrectly from "dirty read" data that was read from an isolation level U plan.

### Locking a Row

If your application needs to hold a lock on a row, you must specify `ISOLEVEL=C` and use the SQL `FETCH` statement to fetch the row using a cursor that has a `WHERE CURRENT OF cursor-name` clause in either an `UPDATE` or `DELETE` statement (the `WHERE CURRENT OF cursor-name` clause need never be executed but, if none are present, blocked transfer of rows may either cause the share lock acquired for `FETCHed` rows to be released before your application fetches them, or a temporary table may be built). The `WHERE CURRENT OF cursor-name` clause causes the row on which your application is positioned to be held with an exclusive lock.

**Note:** Using the SQL `SELECT INTO` statement with `ISOLEVEL=C` causes rows to be accessed by a share lock, but the lock is released before control is passed to your application.

## CICS Unit of Recovery End

If your plan is for a CICS application, CA Datacom CICS Services issues a CICS SYNCPOINT and issues either a COMMIT or ROLBK to end the unit of recovery:

- At the end of each asynchronous CICS task, or
- At the end of each synchronous CICS task (that is, a task associated with a terminal) if:
  - A task abnormally terminates, or
  - A task returns control to CICS, or
  - ISOLEVEL=C and at least one cursor is left open, or
  - ISOLEVEL=C and INSERT, UPDATE, or DELETE statements have been executed.

CA Datacom CICS Services does *not* issue a CICS SYNCPOINT and does *not* issue a COMMIT or ROLBK to end the unit of recovery when:

- A synchronous CICS task terminates successfully, and
- The transaction ID is specified in the RETURN statement, and either:
  - ISOLEVEL=U, or
  - ISOLEVEL=C, no cursor is left open, and no INSERT, UPDATE, or DELETE statements have been executed.

However, the SQL Manager automatically ends a unit of recovery after each SQL statement if there are:

- No open cursors,
- No table-level locks, and
- No primary or secondary exclusive control is being held.

For example, if ISOLEVEL=U and you have only SELECT INTO statements or no open cursors, you need not execute a COMMIT WORK statement at CICS transaction end because there is no current unit of recovery active.

The only case where the unit of recovery is left active when the synchronous CICS task ends successfully is therefore when ISOLEVEL=U and a cursor is left open.

**Note:** If you code ISOLEVEL=U, **you are responsible** for ending the current unit of recovery as follows:

- We recommend ending the current unit of recovery by closing all open cursors. Or, if your user application logic does not support this method,
- Issue a CICS SYNCPOINT (see the *CA Datacom CICS Services User Guide*). Or, if your user application logic does not support this method,
- Issue an appropriate SQL statement (COMMIT WORK or ROLLBACK WORK). Or, if your user application logic does not support this method, as a last option,
- Issue an appropriate CA Datacom/DB record-at-a-time command (LOGCP, LOGCR, LOGTB, COMIT, or ROLBK).

Be aware that if a CA Datacom/DB log command is issued in a situation where a single transaction talks to multiple Multi-User Facilities, unpredictable results can occur. This is why using other methods of ending the current unit of recovery are recommended before this last option.

When you do not do one of the previously given actions and the current unit of recovery remains active, it allows a *browse* application to keep cursors open across CICS synchronous transactions. Be aware, however, that if that was not what you intended, the plan stays locked in share mode, and memory is held in the Multi-User Facility until the unit of recovery is ended.

For example, if your application opens a cursor, fetches one or more rows, and issues a CICS read with return to the application, upon return your application receives a -135 SQLCODE (INVALID CURSOR STATE) when it attempts to open the cursor, because it was left open from the previous CICS transaction. Or, if the next application attempts to execute a plan with ISOLEVEL=C, your application receives a -144 SQLCODE (INVALID TRANSACTION ISOLEVEL) because you cannot mix transaction isolation levels in the same unit of recovery.

To end the current unit of recovery, have your application issue a ROLLBACK WORK or CICS SYNCPOINT when it receives an unexpected SQLCODE, or you can execute a non-SQL command which ends the current unit of recovery. Otherwise, you must close the SQL User Requirements Table to end the current unit of recovery.

If your application does *not* execute CICS RETURN with the *same* transaction ID, the current unit of recovery is ended, that is to say, the unit of recovery is ended if control is passed to another transaction or back to CICS.

## ANSI Compatibility

ISOLEVEL=U is a CA Datacom extension. It is not ANSI standard and is therefore not allowed if ANSI or FIPS is specified in SQLMODE=.



## CA Ideal Considerations

If you are invoking SQL from CA Ideal FOR statements with an embedded TRANSMIT statement, ISOLEVEL=C can be specified. Such cursors are opened in a special mode that holds them open across units of recovery. CA Ideal automatically closes the cursors.

## Block Transfer

If you specify ISOLEVEL=U, block transfer of rows between the Multi-User Facility and the CICS address space is *not* performed.

## OPEN/CLOSE Efficiency

Specifying T for the SQL Preprocessor's PLNCLOSE= option (to close the access plan at unit of recovery end) causes any opened tables to be closed, except when another unit of recovery is currently accessing the table.

If the table is the last table open for an area, the area is "physically" closed, that is to say it is closed by the operating system. We recommend that tables frequently accessed with PLNCLOSE=T be opened in a non-SQL User Requirements Table to avoid this overhead. This User Requirements Table need never be used, but it will keep the areas open.

**Note:** Plan binding uses the CA Datacom Datadictionary, the Schema Information Tables (SIT), and the Optimizer message table (SYSMSG) areas. Binding executes faster if there is a User Requirements Table that holds those areas open. The Temporary Table Manager (TTM) area should also be held open because it too may be used during the execution of binding.

## Automatic Unit of Recovery End

The SQLOPTION Multi-User startup option can be used to automatically close CICS and DLI units of recovery that are left open. For more information on SQLOPTION, see the *CA Datacom/DB Database and System Administration Guide*.

**Note:** A unit of recovery will not be ended if it is still active in the Multi-User Facility (such as when waiting on a plan lock).

If an ISOLEVEL=U application with an open cursor exists for longer than the limit specified in the SQLOPTION Multi-User startup option, the application will receive a -135 SQLCODE (INVALID CURSOR STATE) when it tries to continue scrolling.

Units of recovery that are automatically timed out are not reported.

## Plan Locks

A plan cannot be rebound when in use. If (for a CICS application) R is specified for the SQL Preprocessor's PLNCLOSE= option, the plan remains in use until the SQL User Requirements Table is closed.

To determine which plans are being used, you can use CA Datacom/DB Utility's (DBUTLTY) COMM ALTER option as follows:

### Using COMM ALTER to Determine Plans Being Used

```
►► COMM OPTION=ALTER, TRACE=TRACEGLOBAL ◀◀
```

This writes a report of all open plans (plus various other information about the state of the SQL subsystem) to the CA Datacom/DB Statistics and Diagnostics Area (PXX) when the SQL User Requirements Table is closed. Print the report using CA Datacom/DB Utility's (DBUTLTY) REPORT AREA=PXX option with FULL or TRACE specified for the DUMPS= keyword.

To turn off this option, use CA Datacom/DB Utility's (DBUTLTY) COMM ALTER option:

### Using COMM ALTER to Turn Option Off

```
►► COMM OPTION=ALTER, TRACE=NONE ◀◀
```

**Note:** For more information about using the CA Datacom/DB Utility (DBUTLTY), see the *CA Datacom/DB DBUTLTY Reference Guide*.

### LRU Statement Cache

The LRU (Least Recently Used) statement cache option enhances caching of statements by using a user-specified amount of memory more efficiently. When the LRU option is not used, statements are left in memory until the plan is closed, which can cause the Multi-User Facility to run out of memory by holding on to statements that are rarely re-executed. With the LRU option, however, the amount of memory used is specified by the user and contains only the most recently used statements.

You can set the SQL PLNCLOSE= plan option to close at transaction end or run unit end without concern for caching statements because PLNCLOSE= has no effect on caching when the LRU option is in effect. For example, users who used PLNCLOSE=T for CICS applications have reported a significant decrease in response time by using the LRU option because of reductions in read I/Os to the Data Definition Directory (DDD) database.

Use the console-like command `SQL_LRU_STATEMENT_CACHE n` to activate the LRU statement cache. The value you specify for the `n` parameter controls the size of the cache. The range supported for the `n` parameter is a number from 512000—1073741824.

Use the console-like command `SQL_LRU_STATEMENT_CACHE` to close at transaction end or run unit end.

Statement objects are usually between 5k and 30k, so the 1M default holds approximately 34 to 204 statements.

This option can be changed at any time while the Multi-User Facility is up. If the LRU cache has been on and is then turned off, the virtual storage used by the LRU cache is released. If the LRU cache has been off and it is turned on, the statements cached up to that point in time are released as their plans are closed. But those cached statements will not be used, since only those statements in the LRU cache will be used from this point forward.

To tune the LRU cache by seeing the current size used by the statement cache (either with or without the LRU statement cache option), use the following query:

```
SELECT PLAN_POOL_SIZE
FROM SYSADM.SQL_STATUS;
```

To determine how often a statement is found in the LRU statement cache, turn on the following trace:

```
COMM OPTION=ALTER, TRACE=TRACEGLOBAL, JOBNAME=xxx
```

where `xxx` is the job name of the DBULTY job. This writes general SQL system status information when the DBULTY job ends.

Turn the trace off with the following:

```
COMM OPTION=ALTER, TRACE=NONE
```

The line reporting statement cache performance is as follows:

```
STMT CACHE REQS: n, FOUND n, PERCENT FOUND n%
```

There is no facility to report how often a statement is found in the non-LRU statement cache. However, most requests to the DDD database are usually due to reading statements, so you may use activity to the DDD database as a guide to the effectiveness of statement caching. Be aware that it normally takes from 2 to 9 requests to read each statement from the DDD database when the DDD block size is 4K.



# Chapter 5: Interfacing with the User Requirements Table (URT)

---

Generating User Requirements Interfaces and User Requirements Tables is a database administration activity, which is documented in the *CA Datacom/DB Database and System Administration Guide*. However, the options chosen affect your program, especially a batch program. This chapter describes how these options affect your program.

The User Requirements Interface and Table are generated by assembling the following macros:

## **DBURINF**

CA Datacom/DB Interface

## **DBURSTR**

User Requirements Table Global Parameters

## **DBUREND**

Additional Global Parameters

**Note:** No DBURTBL macros are needed for programs which use (only) SQL to access CA Datacom/DB. The SQL Manager handles the opening and closing of tables referenced by the SQL statements in your program.

## DBURINF - User Requirements Interface

This macro generates the User Requirements Interface for batch programs. For CICS, the interface is generated by the DBCVTPR macro. This macro produces the module DBCSVPR, which is link edited with your program. See CA Datacom CICS Services documentation for details.

In the batch environment, the User Requirements Interface must be link edited with your program, which is described in [Program Compilation, Link-Edit and Execution](#) (see page 281).

**Note:** In z/VSE only, specifying VSERC= in the DBURNIF macro specifies whether a return code is being passed by the application to the operating system in Register 15. If YES is specified, Register 15 is passed to z/VSE unchanged. If NO is specified, Register 15 is ignored by z/VSE. Valid entries for VSERC= are YES or NO. The default is NO.

### Application Program Entry Point

If you want CA Datacom/DB to open and close its environment, your program must be executed as a subroutine of CA Datacom/DB. Your program must be linked using an ENTRY BEGIN control statement. Each language has other special requirements discussed in the following:

#### COBOL

The value of USRNTY= in the DBURINF macro defaults to DBMSCBL, which is the name generated in the source by the Preprocessor. If you want a name other than DBMSCBL, specify that name as the USRNTY= value in the DBURINF macro of the User Requirements Table and for the Preprocessor option of the same name.

#### PL/I

The value of USRNTY= in the DBURINF macro must be the appropriate PL/I entry point, commonly PLISTART.

#### Assembler

The value of USRNTY= in the DBURINF macro must be the program's entry point.

An alternate method is to use the Preprocessor to generate a consistently named entry point. Using this method, a single User Requirements Table assembly may be linked with numerous programs. At preprocess time, the REFNTY= option must be specified. The value of REFNTY= is an existing ENTRY or CSECT in the program. To complete the process, use the USRNTY= option to specify the consistent name. If you do not use the USRNTY= option, a default name, SQLEXECE, is generated.

Your program, regardless of the language, may control the opening and closing of CA Datacom/DB using the language. See the *CA Datacom/DB Database and System Administration Guide* for more information.

## DBURSTR - Start User Requirements Table

The User Requirements Table Start macro defines global program parameters. The DBURSTR macro does not generate assembly output. Instead, it passes its parameters to the User Requirements Table End (DBUREND) macro.

### **MULTUSE=YES**

In the DBURSTR macro, MULTUSE=YES is required (to use SQL, CA Datacom/DB must be operating in Multi-User mode).

## DBUREND - End Interface/Table

The User Requirements Table End macro performs a final edit on the input parameters and generates the assembly output.

### **DBSQL=YES**

In the DBUREND macro, DBSQL=YES is required if your program has SQL statements.

### **z/VSE Load Option**

To use partition GETVIS for loading CA Datacom/DB modules, the DBUREND macro should specify LOADTYP=VIRTUAL.

## Example

In the following Assembler example, note that DBURINF, DBURSTR, DBUREND, and END begin in column 10. The continuation character X is in column 72.

```
TITLE 'DBSBTPR -- BATCH SQL URT'
DBURINF                                     X
    URTABLE=ASM,                           X
    OPEN=DB
DBURSTR                                     X
    MULTUSE=YES,                             X
    TXNUNDO=YES
DBUREND                                     X
    DBSQL=YES,                               X
    USRINFO=SQL-BAT-URT
END
```





# Chapter 6: Program Compilation, Link-Edit and Execution

---

Compile your program according to the standard compilation procedures for your site. After compiling your program, you must link edit the program.

The following section describes batch link-editing and program execution. The next section describes auto-linking between online programs and CA Datacom CICS Services, and online program execution.

## Batch Link-Editing and Execution

When you interface a batch program with the operating system, CA Datacom/DB is mainline and your program is a subroutine of CA Datacom/DB.

- The program is not sensitive to differences in operating system linkages.
- The User Requirements Table can be automatically opened before the program is called, and automatically closed when the program returns control to CA Datacom/DB, or the program can open and close the User Requirements Table.

You must link edit a User Requirements Table (URT) with your program. The DBXHVPR module (host variable processor) must be link edited with your program in addition to the User Requirements Table. DBXHAPR is required if using Dynamic SQL in a COBOL program.

The linkage editor ENTRY BEGIN statement indicates CA Datacom/DB is mainline. CA Datacom/DB calls your program at default entry point DBMSCBL.

### **Batch Program Execution**

After the program has been link edited with the CA Datacom/DB interface, it may be executed.

## Linking Multiple Modules with SQL

Suppose you have the following situation:

- You have a batch COBOL main program named PGMMAIN that *is not* SQL.
- You have a COBOL subprogram named PGMSUB that *is* SQL.
- You preprocess PGMSUB, compile PGMSUB, and catalog the object to a library.

- You compile PGMMAIN (no preprocessing was necessary because it is non-SQL) and link the load module in this order:
  - INCLUDE DBSBTPR (from the library)
  - INCLUDE PGMSUB (from the library)
  - INCLUDE PGMMAIN (from the compiler output)
  - ENTRY BEGIN

**Note:** Use DBSU1PR instead of DBSBTPR if program is AMODE=31 and RMODE=ANY.

Given the previously described situation, the User Requirements Table gets control first, because it has an entry named BEGIN. It then passes control to the entry named DBMSCBL, because that is what the stub URT DBSBTPR (or DBSU1PR) is set up to do. In this scenario, however, the main program is never executed, because CA Datacom/DB passes control directly to the subprogram, and the program abends when (if not before) the subprogram attempts to return to the main program.

To avoid this, you need to design the link-edit so that the CA Datacom/DB User Requirements Table gets control first. The User Requirements Table that is used first then needs to point to the entry point of the main program that is to be executed next. The main program subsequently calls the subprogram, the subprogram returns to the main program, and when the main program ends it returns to the User Requirements Table. After that, the User Requirements Table program finishes and returns control to the operating system.

There are two ways to ensure that the scenario described in the previous paragraph happens:

1. Option 1: You can continue using User Requirements Table DBSBTPR (or DBSU1PR) and compiling and linking programs as before.

The objective here is to make the entry point to which DBSBTPR (or DBSU1PR) is pointing be the entry point of the main program, not the subprogram. To accomplish this, do the following:

- a. In the main program, add ENTRY DBMSCBL at the appropriate point in the code.
- b. If the main program involved has parameters and thus has:

**PROCEDURE DIVISION USING parm1 parm2 ... parm<sub>n</sub>**

the USING clause should be copied onto the ENTRY DBMSCBL so that it becomes:

**ENTRY DBMSCBL USING parm1 parm2 ... parm<sub>n</sub>**

- c. In the subroutine, remove the ENTRY DBMSCBL statement. Replace it with another entry point name. This is done with the SQL preprocessor option USRNTRY=. Specify something other than DBMSCBL. Do not let it default, since DBMSCBL is the default, and we do not want two modules going into the link-edit with the entry point DBMSCBL.
  - d. Link edit the main program, including the User Requirements Table DBSBTPR (or DBSU1PR), the other modules as desired (PGMSUB and PGMMAIN), and ENTRY BEGIN.
2. Option 2: Create another stub User Requirements Table instead of using DBSBTPR (or DBSU1PR).

The objective here is to create another stub User Requirements Table that points to an entry point (other than DBMSCBL) in the main program. To accomplish this, do the following:

- a. Create a User Requirements Table with no files specified. Choose USRNTRY= to be the appropriate entry point of the main program. See the *CA Datacom/DB Database and System Administration Guide* for instructions on creating User Requirements Table's.
- b. Link edit the main program, including the new User Requirements Table (using whatever name you chose), other modules as desired (PGMSUB and PGMMAIN), and ENTRY BEGIN.

Remember, *if* the call to the SQL subprogram is being added to the main program at this point, *and* the main program does not use DATACOM, the "main" program is actually now running as a subprogram to the User Requirements Table program, and you should therefore use GOBACK instead of STOP RUN in the main program.

**Note:** A benefit of using option 1 for situations such as this is that you can probably use standard link-edit JCL for all programs. Option 2 requires the use of another stub User Requirements Table, either one such User Requirements Table per main non-SQL program that calls SQL subprograms, or one stub User Requirements Table for all such non-SQL main programs that call SQL subprograms. Also with option 2, the INCLUDE statements have to be different in the link-edit jobs if different stub User Requirements Table's were used.

## Sample JCL for Batch

### **COBOL**

For COBOL samples see [z/OS Sample COBOL JCL for Batch \(z/OS\)](#) (see page 152) or [z/VSE Sample COBOL JCL for Batch \(z/VSE\)](#) (see page 157).

### **PL/I**

For PL/I samples see [z/OS PL/I Sample JCL \(z/OS\)](#) (see page 161) or [z/VSE PL/I Sample JCL \(z/VSE\)](#) (see page 163).

### **Assembler**

For Assembler samples see [z/OS Assembler Sample JCL \(z/OS\)](#) (see page 164) or [z/VSE Assembler Sample JCL. \(z/VSE\)](#) (see page 167).

### **C Language**

See [Sample C Language JCL](#) (see page 168).

## CICS Link-Editing and Execution

Programs running under the control of IBM's teleprocessing monitor CICS are subroutines to CICS. They access CA Datacom/DB using CA Datacom CICS Services.

The module DBCSVPR is link edited with the program to provide linkage between the program and CA Datacom CICS Services.

The DBXHVPR module for the host variable processor must also be link edited. DBXHAPR is required if using Dynamic SQL in a COBOL program.

An include card may be used to include these modules, as in the following example:

```
INCLUDE SYSLIB(DBCSVPR)
INCLUDE OBJLIB(DBXHVPR)
NAME modname
```

CA Datacom CICS Services opens and closes all User Requirements Tables.

### **Online Program Execution**

Follow the procedure for the online monitor you are using. There are no special execution rules for programs using CA Datacom/DB.

## Sample JCL for CICS

### **z/OS CICS**

For sample z/OS CICS JCL see [z/OS Sample COBOL JCL for CICS](#) (see page 154).

### **z/VSE CICS**

For sample z/VSE CICS JCL see [z/VSE Sample COBOL JCL for CICS](#) (see page 159).

## IMS/DC Link-Editing and Execution

Programs running under the control of IBM's teleprocessing monitor IMS/DC are subroutines to IMS. They access CA Datacom/DB using CA Datacom IMS/DC Services. The CA Datacom language interface DFSLI000 is link edited with the program to provide linkage between the program and CA Datacom IMS/DC Services. The DBXHVPR module for the host variable processor must also be link edited. DBXHAPR is required if using Dynamic SQL in a COBOL program. The control cards for the link-edit step are:

```
INCLUDE SYSLIB(DBXHVPR)
INCLUDE IMSDCLIB(CBLTDLI)
ENTRY DLITCBL
NAME PROGRAM(R)
```

CA Datacom IMS/DC Services opens and closes all User Requirements Tables. The User Requirements Tables are not linked with the program. SYSLIB is your CA Datacom load library. IMSDCLIB is the CA Datacom IMS/DC Services library where the language interface DFSLI000 resides.

**Note:** For more information, see the section on using SQL with IMS/DC in the *CA Datacom IMS/DC Services System Guide*.

## Sample JCL for IMS/DC

The following example shows z/OS JCL for precompile, compile, and link. This example is intended as a sample only. You need to modify the JCL to conform to the requirements of your site..

## z/OS IMS/DC Sample JCL

The following z/OS JCL example is for programs running under CA Datacom IMS/DC Services.

**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.
//*****
//* THE FOLLOWING JOB STREAM DEMONSTRATES THE SQL
//* PREPROCESSOR, THE CICS COMMAND LEVEL PREPROCESSOR AND THE COBOL
//* COMPILER STEPS
//*****
//STEP1     EXEC PGM=DBXMMPR
//STEPLIB   See the note above
//WORK1     DD DSN=&.&WORK1. ,UNIT=SYSDA,DISP=(NEW,PASS) ,
//           DCB=(RECFM=F,LRECL=80,BLKSIZE=80) ,SPACE=(TRK,(1,1))
//WORK2     DD DSN=&.&WORK2. ,UNIT=SYSDA,DISP=(NEW,PASS) ,
//           DCB=(RECFM=F,LRECL=80,BLKSIZE=80) ,SPACE=(TRK,(1,1))
//WORK3     DD DSN=&.&WORK3. ,UNIT=SYSDA,DISP=(NEW,PASS) ,
//           DCB=(RECFM=F,LRECL=80,BLKSIZE=80) ,SPACE=(TRK,(1,1))
//SYSOUT    DD SYSOUT=*
//SYSPRINT  DD SYSOUT=*                               Print Output
//SYSPUNCH  DD DSN=&.&TEMP. ,UNIT=SYSDA,DISP=(NEW,PASS) ,
//           DCB=(RECFM=FB,LRECL=80,BLKSIZE=800) ,SPACE=(TRK,(2,1))
//SYSUDUMP  DD SYSOUT=*
//SNAPER    DD SYSOUT=*
//INCLUDE   DD DSN=ca.user.include.library,DISP=SHR
//SYSIN     DD *                                       Command input

        PLACE COBOL SOURCE TEXT HERE.
//*****
//* COBOL COMPILER STEP
//*****
//COB       EXEC PGM=IKFCBL00,REGION=1024K,
//           PARM='NOTRUNC,NODYNAM,LIB,SIZE=1024K,BUF=16K' ,
//           COND=(4,GT)
//SYSLIB    DD DSN=CICS.COBLIB,DISP=SHR
//           DD DSN=SYS1.COBOLINK,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD DSN=&.&SYSCIN. ,DISP=(OLD,DELETE)
//SYSLIN    DD DSN=&.&LOADSET. ,DISP=(MOD,PASS) ,
//           UNIT=DISK,SPACE=(80,(250,100))
//SYSUT1    DD UNIT=DISK,SPACE=(460,(350,100))
//SYSUT2    DD UNIT=DISK,SPACE=(460,(350,100))
//SYSUT3    DD UNIT=DISK,SPACE=(460,(350,100))
```

```

//*****
//*      LINK EDIT STEP
//*****
//LKED      EXEC PGM=IEWL,REGION=1024K,PARM=XREF,COND=(4,GT)
//SYSLIB    DD DSN=ca.cobol.compiler.loadlib,DISP=SHR
//          DD DSN=ca.datacom.loadlib,DISP=SHR
//IMSDCLIB DD DSN=yourimsdclib,DISP=SHR
//SYSLMOD   DD DSN=ca.user.loadlib,DISP=SHR
//SYSUT1    DD UNIT=DISK,DCB=BLKSIZE=1024,
//          SPACE=(1024,(200,20))
//SYSPRINT  DD SYSOUT=*
//SYSLIN    DD DSN=&.&LOADSET.,DISP=(OLD,DELETE)
//          DD *
INCLUDE IMSDCLIB(CBLTDLI)
INCLUDE SYSLIB(DBXHVPR)
ENTRY DLITCBL
NAME TEST01(R)
/*
//

```





# Chapter 7: SQL Error Handling

---

If you are using the CA Datacom/DB SQL Preprocessor for COBOL or the Interactive SQL Service Facility, errors can generate non-SQL return codes or messages, as described in the table and the following information.

**Note:** For a complete list of SQL return codes, see the *CA Datacom/DB Message Reference Guide*.

The SQL Manager passes a value to the SQLCODE field of the SQL Communications Area after each SQL statement is processed during preprocessing or during program execution. Most values for the SQLCODE indicate that the error was detected within the SQL Manager. Two special cases occur when SQLCODE contains -117 or -118 (see [SQL Return Codes -117 and -118](#) (see page 289)).

Beginning with r11, a SQLSTATE status indicator is listed with each SQLCODE. See the SQL return code information in the *CA Datacom/DB Message Reference Guide* and [SQL States](#) (see page 298) in this chapter..

## SQL Return Codes -117 and -118

If SQLCODE contains -117, the error was detected by CA Datacom/DB. Check other SQLCA fields for the internal and/or external CA Datacom/DB return code to determine what action you should take. See the *CA Datacom/DB Message Reference Guide* for descriptions of CA Datacom/DB return codes and messages.

If SQLCODE contains -118, the error was detected by the CA Datacom Datadictionary Service Facility. Check other SQL Communication Area (SQLCA) fields for a specific CA Datacom Datadictionary Service Facility return code to determine what caused the error. See the *CA Datacom/DB Message Reference Guide* for descriptions of CA Datacom Datadictionary Service Facility return codes and CA Datacom Datadictionary messages.

The following table lists the information placed in other SQLCA fields. Code your program so that it prints or displays all of the SQLCA's return code information.

SQLCA Field	SQLCODE=0 No Error	SQLCODE=-117 CA DATACOM/DB	SQLCODE=-118 CA Datacom Datadictionary Service Facility
SQLCA-DSFCODE	bbbb		Naaa Mnnn Mccc D000

SQLCA Field	SQLCODE=0 No Error	SQLCODE=-117 CA DATACOM/DB	SQLCODE=-118 CA Datacom Datadictionary Service Facility	
SQLCA-DBCODE-INT	null	n	null	n
SQLCA-DBCODE-EXT	bb	nn	bb	nn
SQLCA-ERROR-PGM	b	DBSERV	b	ccc DDL
SQLCA-ERR-MSG	b	error message	return data	return data
SQLSTATE	00000	Seeii	Rdddd	Reeii

**b**

A blank indicates the command was successful, or no data was returned to the field.

**null**

A null value (binary zeros).

**n**

The one-byte CA Datacom/DB internal return code. See the section on CA Datacom/DB Return Codes in the *CA Datacom/DB Message Reference Guide* for a description of the error and corrective action to take.

**nn**

The two-byte CA Datacom/DB external return code. See the section on CA Datacom/DB Return Codes in the *CA Datacom/DB Message Reference Guide* for a description of the error and corrective action to take.

**DBSERV**

Internal information.

**Naaa**

The *N* indicates this is a normal alphanumeric return code, while the *aaa* represents the actual alphanumeric return code from the CA Datacom Datadictionary Service Facility. See the section on CA Datacom Datadictionary Service Facility Return Codes in the *CA Datacom/DB Message Reference Guide* for a description of the error and corrective action to take.

**Mnnn**

*M* indicates return code is from a module of CA Datacom Datadictionary. The *nnn* represents the numeric return code from the CA Datacom Datadictionary Service Facility. Use the first three bytes of the SQLCA-ERROR-PGM field as the first three three of the return code. See the section on CA Datacom Datadictionary Service Facility Internal Return Codes in the *CA Datacom/DB Message Reference Guide* for a description of the error and corrective action to take.

**Mccc**

The *M* indicates the return code is from a module of CA Datacom Datadictionary, while the *ccc* represents the CA Datacom Datadictionary Service Facility return code from the interface module that populates CA Datacom Datadictionary. See the section on CA Datacom Datadictionary Service Facility Internal Return Codes in the *CA Datacom/DB Message Reference Guide* for a description of the error and corrective action to take.

**D000**

Indicates that a CA Datacom/DB return code was received during CA Datacom Datadictionary Service Facility processing. See the SQLCA-DBCDEF for the CA Datacom/DB internal and external return codes.

**ccc**

Indicates the CA Datacom Datadictionary Service Facility module reporting the error. This three-character module name and the last three bytes of the SQLCA-DSFCODE field indicate the error. See the *CA Datacom/DB Message Reference Guide* for a description of the error and corrective action to take.

**DDL**

Indicates the error was reported by the interface module that populates CA Datacom Datadictionary.

**return data**

If applicable, up to 80 bytes of formatted information specific to the request causing the error is returned in the SQLCA-ERR-MSG field. Check the SQLCA-DSFCODE field for the return code and see the *CA Datacom/DB Message Reference Guide* for a description of the error and corrective action to take.

If the SQLCA-DSFCODE field contains D000, the information in the return data indicates the CA Datacom/DB request issued when the error was detected, including the command, the return code and the CA Datacom/DB table name.

**Rdddd**

The R in the first position associates the SQLSTATE with -118. The dddd is the DSF code returned to SQL.

**Reeii**

The R in the first position associates the SQLSTATE with -118. The ee represents the 2-byte external CA Datacom/DB return code. The ii represents the CA Datacom/DB internal return code in hexadecimal characters.

**Seeii**

The S in the first position associates the SQLSTATE with -117. The ee represents the 2-byte external CA Datacom/DB return code. The ii represents the CA Datacom/DB internal return code in hexadecimal characters.

## Online Displays

The following examples show how -117 and -118 are displayed on CA Datacom Datadictionary Interactive SQL Service Facility panels when an error is detected when executing an SQL statement.

**Example 1 (SQLCODE = -117)**

```

=>
=>
=>
----->>>
Interactive SQL Service Facility                               SQLMAINT
                SQL Output Panel                               S010
          EDIT      Member: $DDSQL
                Description: CREATE TABLE FOR DEPTTBL
-----
000010 NUMBER OF SQL STATEMENTS PROCESSED IS 0001
000011 (OUTPUT CREATED FROM SQL SOURCE MEMBER $DDSQL)
000012 create table depttbl
000013     (deptno char (2),
000014     deptname char (24));
000015 -----+-----+-----+-----+-----+-----+
000016 ERROR OCCURRED DELETING A PLAN -0117(36,192)          (QDELP)
000017 DB ERROR OCCURRED DURING SQL PROCESSING
000018 -----+-----+-----+-----+-----+-----+
===== B O T T O M =====
PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT    PF12=ALTERNATE

```

In this panel, the error messages and return codes that display are as follows:

ERROR OCCURRED ccccccccccccccccccc snnnn(xx,yyy) (bbbb)

Indicates an error was encountered during SQL statement execution.

**cccccccccccccccccccc**

Indicates where the error occurred during the SQL statement execution. In the previous example, an error occurred deleting a plan.

**snnnn(xx,yyy)**

The snnnn string indicates that an error occurred during CA Datacom/DB or CA Datacom Datadictionary execution, where the s is the sign (+ or -) and nnnn is the error code number. The value -0117 of snnnn in the previous example indicates an error occurred in executing CA Datacom/DB.

The (xx,yyy) represents the external and internal CA Datacom/DB return codes where xx is the external return code and yyy is the internal return code.

**(bbbb)**

These characters identify the internal CA Datacom/DB SQL command.

**Example 2 (SQLCODE = -118)**

```

=>
=>
=>

----->>>
Interactive SQL Service Facility                               SQLMAINT

                SQL Output Panel                               S010
          EDIT      Member: $DDSQL
                Description: CREATE MY SCHEMA
-----
000001 (OUTPUT CREATED FROM SQL SOURCE MEMBER $DDSQL)
000002 create schema authorization jones;
000003 -----+-----+-----+-----+-----+-----+-----+
000004 ERROR OCCURRED EXECUTING A PLAN  -0118(MAAE)      DDLGETEN (QEXEI)
000005 -----+-----+-----+-----+-----+-----+-----+
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT    PF12=ALTERNATE
  
```

In this panel, the error messages and return codes that display are as follows:

```
ERROR OCCURRED ccccccccccccccccccc snnnn(aaaa :xx,yyy) dddddddd (bbbb)
```

Indicates an error was encountered during SQL statement execution.

**cccccccccccccccccccc**

Indicates where the error occurred during the SQL statement execution. In the previous example, an error occurred executing a plan.

**snnnn(aaaa :xx,yyy)**

The snnnn string indicates that an error occurred during CA Datacom/DB or CA Datacom Datadictionary execution, where s is the sign (+ or -) and nnnn is the error number. The value -0118 of snnnn in the previous example indicates an error occurred in executing CA Datacom Datadictionary.

The aaaa string indicates the CA Datacom Datadictionary Service Facility return code and corresponds to the SQLCA-DSFCODE. (See the table at the start of this chapter for the interpretation of this error.) The M in MAAE in the previous example means that the return code is from a module of CA Datacom Datadictionary, while the three characters following the M are the CA Datacom Datadictionary Service Facility return code from the interface module that populates CA Datacom Datadictionary. In this case, AAE means "authorization already exists."

When aaaa is D000, xx,yyy represents the external and internal CA Datacom/DB return codes where xx is the external return code and yyy is the internal return code. In the previous example, no CA Datacom/DB return code is displayed on the panel since the error was not one detected by CA Datacom/DB.

**ddddddd**

The DDL in DDLGETEN in the previous example identifies the module that populates CA Datacom Datadictionary, while the GETEN is a CA Datacom Datadictionary Service Facility command.

**(bbbb)**

These characters identify the internal CA Datacom/DB SQL command.

**Example 3 (SQLCODE = 243)**

```

=>
=>
=>

----->>>
Interactive SQL Service Facility                                SQLMAINT

                SQL Output Panel                                S010
            EDIT      Member: $DDSQL
                Description: DROP TABLE DEPTTBL
-----
000010 NUMBER OF SQL STATEMENTS PROCESSED IS 0001
000011 (OUTPUT CREATED FROM SQL SOURCE MEMBER $DDSQL)
000012 drop table depttbl;
000013 -----+-----+-----+-----+-----+-----+-----+
000014 ERROR OCCURRED EXECUTING A PLAN +0243          TBLUPD (QEXEI)
000015 ANSI EXTENSION
000016 -----+-----+-----+-----+-----+-----+
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT    PF12=ALTERNATE

```

In this panel, the error messages and return codes that display are as follows:

ERROR OCCURRED ccccccccccccccccccc snnn(xx,yyy) aaaaaa (bbbb)

Indicates an error was encountered during SQL statement execution.

**cccccccccccccccccccc**

Indicates where the error occurred during the SQL statement execution. In the previous example, an error occurred executing a plan.

**snnn(xx,yyy)**

The snnn string indicates that an error occurred during CA Datacom/DB or CA Datacom Datadictionary execution, where the s is the sign (+ or -) and nnnn is the error code number.

**Note:** In the list of SQL codes found in the *CA Datacom/DB Message Reference Guide*, positive return codes are listed without the plus (+) sign and without the leading zero, for example the previously shown +0243, is listed as 243.

The (xx,yyy) represents the external and internal CA Datacom/DB return codes where xx is the external return code and yyy is the internal return code.

**aaaaaa**

These characters identify the function being performed.

**(bbbb)**

These characters identify the internal CA Datacom/DB SQL command.

## Batch Output

The following examples show how an error handling routine can print pertinent return code information when SQLCODE contains -117 or -118.

Following is the COBOL error routine used to output these examples:

ERRORTN.

```
MOVE SQLCODE TO WK-CODE.
IF (SQLCODE IS LESS THAN 0)
  DISPLAY 'SQLCODE =' WK-CODE '(MINUS) '
ELSE
  DISPLAY 'SQLCODE =' WK-CODE.
DISPLAY 'ERRPGM =' SQLCA-ERROR-PGM.
DISPLAY 'ERRMSG =' SQLCA-ERR-MSG.
DISPLAY 'SQLCA-DSFCODE=' SQLCA-DSFCODE.
MOVE SQLCA-INFCODE TO WK-CODE.
DISPLAY 'SQLCA-INFCODE=' WK-CODE.
DISPLAY 'SQLCA-DBCOD-EXT =' SQLCA-DBCOD-EXT.
MOVE SQLCA-DBCOD-INT TO WK-CODE.
DISPLAY 'SQLCA-DBCOD-INT =' WK-CODE.
DISPLAY 'SQLERR4 =' SQLCA-MISC-CODE1.
DISPLAY 'SQLERR5 =' SQLCA-MISC-CODE2.
DISPLAY 'SQLERR6 =' SQLCA-MISC-CODE3.
DISPLAY 'SQLWARN1=' SQLCA-WARNING(1).
DISPLAY 'SQLWARN2=' SQLCA-WARNING(2).
DISPLAY 'SQLWARN3=' SQLCA-WARNING(3).
DISPLAY 'SQLWARN4=' SQLCA-WARNING(4).
DISPLAY 'SQLWARN5=' SQLCA-WARNING(5).
DISPLAY 'SQLWARN6=' SQLCA-WARNING(6).
DISPLAY 'SQLWARN7=' SQLCA-WARNING(7).
DISPLAY 'SQLWARN8=' SQLCA-WARNING(8).
DISPLAY 'SQLSTATE=' SQLSTATE.
MOVE +4 TO RETURN-CODE.
GOBACK.
```



**Example 1 (SQLCODE = -117)**

In the following example, the -117 indicates the error was detected by CA Datacom/DB, so appropriate return code information is included in the output.

```
SQLCODE =0117(MINUS)
ERRPGM  =DBSERV
ERRMSG  =DB ERROR OCCURRED DURING SQL PROCESSING
SQLCA-DSFCODE=
SQLCA-INFICODE= 0000
SQLCA-DBCOD-EXT =09
SQLCA-DBCOD-INT= 0037
SQLERR4 =
SQLERR5 =00000000J
SQLERR6 =
SQLWARN1=
SQLWARN2=
SQLWARN3=
SQLWARN4=
SQLWARN5=
SQLWARN6=
SQLWARN7=
SQLWARN8=
SQLSTATE=S0925
```

**Example 2 (SQLCODE = -118)**

In the following example, the -118 indicates the error was detected by CA Datacom Datadictionary, so appropriate return code information is included in the output.

```
SQLCODE =0118(MINUS)
ERRPGM  =DDLGETEN
ERRMSG  =
SQLCA-DSFCODE= MANF
SQLCA-INFICODE= 0000
SQLCA-DBCOD-EXT =
SQLCA-DBCOD-INT= 0000
SQLERR4 =
SQLERR5 =00000000J
SQLERR6 =
SQLWARN1=
SQLWARN2=
SQLWARN3=
SQLWARN4=
SQLWARN5=
SQLWARN6=
SQLWARN7=
SQLWARN8=
SQLSTATE=RMANF
```

## Error Handling Related to Procedures and Triggers

For information about error handling related to procedures and triggers, see [Parameter Styles and Error Handling](#) (see page 80).

Also see [SQL Error Messages Related to Procedures and Triggers](#) (see page 83).

## SQL States

Prior to r11, the SQLCODE was the sole indicator to users of the success or failure of an SQL transaction. Beginning with r11, in addition to the SQLCODE a SQLSTATE status indicator is provided that corresponds to any error or completion condition.

No special configuration is needed to use the SQLSTATE status indicators. Applications can use the SQLSTATE simply by referencing the new SQLSTATE field.

When a procedure is created using PARAMETER STYLE SQL in the CREATE PROCEDURE statement, the SQLSTATE is returned in the SQLCA.

The SQLSTATE feature is backward-compatible with existing applications. because the fields holding the SQLSTATE in the SQLCA were reserved fields in prior versions. However, any existing, embedded SQL program that has defined a field named SQLSTATE fails any new preprocessing because of the duplicate name.

Following are the layouts of the non-DB2 mode SQLCA format in COBOL, PL/I, Assembler, and C (note boldface lines in the examples).

**Note:** The DB2 mode SQLCA formats are unchanged from prior versions.

## SQLCA Examples

Examples follow of the SQLCA in the following language formats:

- COBOL (see [SQLCA - CA Datacom/DB Format \(COBOL\)](#) (see page 299))
- PL/I (see [SQLCA - CA Datacom/DB Format \(PL/I\)](#) (see page 300))
- Assembler (see [SQLCA - CA Datacom/DB Format \(Assembler\)](#) (see page 303))
- C (see [SQLCA in C Language](#) (see page 304))

## SQLCA - CA Datacom/DB Format (COBOL)

```

01 SQLCA.
  01 SQLCA.
    05 SQLCA-EYE-CATCH          PIC X(08).
    05 SQLCAID REDEFINES SQLCA-EYE-CATCH
                                   PIC X(08).
    05 SQLCA-LEN                PIC S9(9) COMP.
    05 SQLCABC REDEFINES SQLCA-LEN
                                   PIC S9(9) COMP.
    05 SQLCA-DB-VRS             PIC X(02).
    05 SQLCA-DB-RLS             PIC X(02).
    05 SQLCA-LUWID              PIC X(08).
    05 SQLCODE                  PIC S9(9) COMP.
    05 SQLCA-ERROR-INFO.
      10 SQLCA-ERR-LEN          PIC S9(4) COMP.
      10 SQLCA-ERR-MSG         PIC X(80).
    05 SQLERRM REDEFINES SQLCA-ERROR-INFO.
      10 SQLERRML              PIC S9(4) COMP.
      10 SQLERRMC              PIC X(70).
    05 SQLCA-ERROR-PGM         PIC X(08).
    05 SQLERRP REDEFINES SQLCA-ERROR-PGM
                                   PIC X(08).
    05 SQLCA-FILLER-1          PIC X(02).
    05 SQLCA-ERROR-DATA.
      10 SQLCA-DSFCODE         PIC X(04).
      10 SQLCA-INFCODE         PIC S9(9) COMP.
      10 SQLCA-DBCOD.
        15 SQLCA-DBCOD-EXT PIC X(02).
        15 SQLCA-DBCOD-INT PIC S9(4) COMP.
      10 SQLCA-MISC-DATA.
        15 SQLCA-MISC-CODE2 PIC S9(9) COMP.
        15 SQLCA-MISC-CODE3 PIC S9(9) COMP.
      10 SQLCA-ERR-DIAG REDEFINES SQLCA-MISC-DATA.
        15 SQLSTATE           PIC X(05).
        15 SQLCA-FILLER-2     PIC X(03).
    05 SQLCA-WRN-AREA.
      10 SQLCA-WARNING         PIC X OCCURS 8 TIMES.

```

```

05  SQLWARN REDEFINES SQLCA-WRN-AREA.
    10  SQLWARN0          PIC X.
    10  SQLWARN1          PIC X.
    10  SQLWARN2          PIC X.
    10  SQLWARN3          PIC X.
    10  SQLWARN4          PIC X.
    10  SQLWARN5          PIC X.
    10  SQLWARN6          PIC X.
    10  SQLWARN7          PIC X.
05  SQLCA-PGM-NAME
     PIC X(08).
05  SQLCA-AUTHID
     PIC X(18).
05  SQLCA-PLAN-NAME
     PIC X(18).

```

**Note:** All REDEFINES are for compatibility with other SQL implementations.

## SQLCA - CA Datacom/DB Format (PL/I)

```

DCL 1 SQLCA,
    5 SQLCA_EYE_CATCH   CHAR(8) INIT('SQLCA***'),
    5 SQLCA_LEN         FIXED BINARY(31) INIT(196),
    5 SQLCA_DB_VRS     CHAR(2) INIT('08'),
    5 SQLCA_DB_RLS     CHAR(2) INIT('10'),
    5 SQLCA_LUWID      CHAR(8) INIT(' '),
    5 SQLCA_CODE       FIXED BINARY(31),
    5 SQLCA_ERR_LEN    FIXED BINARY(15),
    5 SQLCA_ERR_MSG    CHAR(80) INIT(' '),
    5 SQLCA_ERROR_PGM  CHAR(8) INIT(' '),
    5 SQLCA_FILLER_1   CHAR(2) INIT(' '),
    5 SQLCA_DSFCODE    CHAR(4) INIT(' '),
    5 SQLCA_INFPCODE   FIXED BINARY(31),
    5 SQLCA_DBCODE_EXT CHAR(2) INIT(' '),
    5 SQLCA_DBCODE_INT FIXED BINARY(15),
    5 SQLCA_MISC_CODE1 FIXED BINARY(31),
    5 SQLCA_MISC_CODE2 FIXED BINARY(31),
    5 SQLCA_MISC_CODE3 FIXED BINARY(31),
    5 SQLCA_WRN_AREA,
    10 SQLCA_WARNING (0:7) CHAR(1) INIT(' '),
    5 SQLCA_PGM_NAME   CHAR(8) INIT(' '),
    5 SQLCA_AUTHID     CHAR(18) INIT('authid here '),
    5 SQLCA_PLAN_NAME  CHAR(18) INIT('plan name here ');

```

```
DCL SQLCAID          CHAR(8)
                     DEFINED SQLCA_EYE_CATCH;
DCL SQLCABC          FIXED BINARY(31)
                     DEFINED SQLCA_LEN;
DCL SQLERRML         FIXED BINARY(15)
                     DEFINED SQLCA_ERR_LEN;
DCL SQLERRMC         CHAR(80)
                     DEFINED SQLCA_ERR_MSG;
DCL SQLERRP          CHAR(8)
                     DEFINED SQLCA_ERROR_PGM;
DCL 1 SQLWARN        DEFINED SQLCA_WRN_AREA,
    5 SQLWARN0        CHAR(1),
    5 SQLWARN1        CHAR(1),
    5 SQLWARN2        CHAR(1),
    5 SQLWARN3        CHAR(1),
    5 SQLWARN4        CHAR(1),
    5 SQLWARN5        CHAR(1),
    5 SQLWARN6        CHAR(1),
    5 SQLWARN7        CHAR(1);
DCL 1 SQLCA_WARN     DEFINED SQLCA_WRN_AREA,
    5 SQLCA_WARN0     CHAR(1),
    5 SQLCA_WARN1     CHAR(1),
    5 SQLCA_WARN2     CHAR(1),
    5 SQLCA_WARN3     CHAR(1),
    5 SQLCA_WARN4     CHAR(1),
    5 SQLCA_WARN5     CHAR(1),
    5 SQLCA_WARN6     CHAR(1),
    5 SQLCA_WARN7     CHAR(1);
```

**Note:** All REDEFINES are for compatibility with other implementations of SQL.

```
DCL 1 SQLCA,
  5 SQLCA_EYE_CATCH      CHAR(8) INIT('SQLCA***'),
  5 SQLCA_LEN FIXED      BINARY(31) INIT(196),
  5 SQLCA_DB_VRS         CHAR(2) INIT('08'),
  5 SQLCA_DB_RLS         CHAR(2) INIT('10'),
  5 SQLCA_LUWID          CHAR(8) INIT(' '),
  5 SQLCODE              FIXED BINARY(31),
  5 SQLCA_ERR_LEN        FIXED BINARY(15),
  5 SQLCA_ERR_MSG        CHAR(80) INIT(' '),
  5 SQLCA_ERROR_PGM      CHAR(8) INIT(' '),
  5 SQLCA_FILLER_1       CHAR(2) INIT(' '),
  5 SQLCA_DSFCODE         CHAR(4) INIT(' '),
  5 SQLCA_INFCD           FIXED BINARY(31),
  5 SQLCA_DBCODE_EXT     CHAR(2) INIT(' '),
  5 SQLCA_DBCODE_INT     FIXED BINARY(15),
  5 SQLCA_MISC_CODE1     FIXED BINARY(31),
  5 SQLCA_MISC_DATA      CHAR(8),
  5 SQLCA_WRN_AREA,
  10 SQLCA_WARNING (0:7) CHAR(1) INIT(' '),
  5 SQLCA_PGM_NAME       CHAR(8) INIT(' '),
  5 SQLCA_AUTHID         CHAR(18) INIT('authid here '),
  5 SQLCA_PLAN_NAME      CHAR(18) INIT('plan name here ');
DCL SQLCAID              CHAR(8)          DEF SQLCA_EYE_CATCH;
DCL SQLCABC              FIXED BINARY(31) DEF SQLCA_LEN;
DCL SQLERRML             FIXED BINARY(15) DEF SQLCA_ERR_LEN;
DCL SQLERRMC             CHAR(80)         DEF SQLCA_ERR_MSG;
DCL SQLERRP              CHAR(8)         DEF SQLCA_ERROR_PGM;
DCL SQLCA_SQLSTATE       CHAR(5)         DEF SQLCA_MISC_DATA,
  SQLSTATE               CHAR(5)         DEF SQLCA_MISC_DATA;
DCL 1 SQLWARN            DEFINED SQLCA_WRN_AREA,
  5 SQLWARN0             CHAR(1),
  5 SQLWARN1             CHAR(1),
  5 SQLWARN2             CHAR(1),
  5 SQLWARN3             CHAR(1),
  5 SQLWARN4             CHAR(1),
  5 SQLWARN5             CHAR(1),
  5 SQLWARN6             CHAR(1),
  5 SQLWARN7             CHAR(1);
```

```

DCL 1 SQLCA_WARN          DEFINED SQLCA_WRN_AREA,
      5 SQLCA_WARN0      CHAR(1),
      5 SQLCA_WARN1      CHAR(1),
      5 SQLCA_WARN2      CHAR(1),
      5 SQLCA_WARN3      CHAR(1),
      5 SQLCA_WARN4      CHAR(1),
      5 SQLCA_WARN5      CHAR(1),
      5 SQLCA_WARN6      CHAR(1),
      5 SQLCA_WARN7      CHAR(1);

```

**Note:** All REDEFINES are for compatibility with other implementations of SQL.

### SQLCA - CA Datacom/DB Format (Assembler)

```

SQLCA  DSECT
SQLCAEYE DS  CL8  .EYE CATCHER
SQLCALEN DS  F    .BLOCK LENGTH
SQLCADBV DS  CL2  .DB VERSION
SQLCADBR DS  CL2  .DB RELEASE
SQLCALIW DS  CL8  .LUW ID
SQLCODE DS  F    .SQL RETURN CODE
SQLCAERI DS  0CL82 .ERROR TEXT
SQLCAELN DS  H    . LENGTH
SQLCAEMS DS  CL80 . MESSAGE
SQLCAEPG DS  CL8  . PROGRAM
SQLCAFL1 DS  CL2  .UNUSED
SQLCAEDT DS  0CL22 .ERROR DATA
SQLCADSF DS  CL4  . DSF EXTERNAL CODE
SQLCAINF DS  F    . RESERVED
SQLCADBC DS  0CL6 . DB CODES
SQLCADBX DS  CL2  . EXTERNAL

```

SQLCADBI	DS	H	.	INTERNAL
SQLCAMC1	DS	F	.	ROWS AFFECTED
SQLSTATE	DS	0CL5	.	SQLSTATE
SQLCAMC2	DS	F	.	RESERVED
SQLCAMC3	DS	F	.	RESERVED
SQLCAWRN	DS	0CL8	.	WARNINGS
SQLCAWN0	DS	CL1	.	SQLCA WARNING
SQLCAWN1	DS	CL1	.	RESERVED
SQLCAWN2	DS	CL1	.	RESERVED
SQLCAWN3	DS	CL1	.	UNEQUAL VARS
SQLCAWN4	DS	CL1	.	RESERVED
SQLCAWN5	DS	CL1	.	DATE/TIMESTAMP ADJUSTMENT
SQLCAWN6	DS	CL1	.	RESERVED
SQLCAWN7	DS	CL1	.	RESERVED
SQLCAPGM	DS	CL8	.	UNUSED
SQLCAATH	DS	CL18	.	AUTH ID
SQLCAPLN	DS	CL18	.	PLAN NAME
SQLCADLN	EQU	*	-	SQLCA

**Note:** All REDEFINES are for compatibility with other implementations of SQL.

## SQLCA in C Language

```
struct sqlca {
    char    sqlca_eye_catch [8];
    #define sqlcaid    sqlca_eye_catch
    int     sqlca_len;
    #define sqlcabc    sqlca_len

    #ifndef DB2
        char    sqlca_db_vrs    [2];
        char    sqlca_db_rls    [2];
        char    sqlca_lwid      [8];
    #endif

    int     sqlca_code;
    #define sqlcode    sqlca_code
    #define sqlcade    sqlca_code

    #ifndef DB2
        short   sqlca_err_len;
        #define sqlerrml    sqlca_err_len
    #endif

    char    sqlca_err_msg SQLCA_MSG_LEN&hyphen. ;
    #define sqlerrmc    sqlca_err_msg
    #define sqlerrm    sqlca_err_msg
    char    sqlca_error_pgm    [8];
    #define sqlerrp    sqlca_error_pgm
}
```



```

#ifdef DB2
    int    sqlerrd 6&hyphen. ;
#else
    char   sqlca_filler_1  [2];
    char   sqlca_dsfcodes  [4];
    int    sqlca_infcode;
    char   sqlca_dbcode_ext [2];
    short  sqlca_dbcode_int;
    int    sqlca_misc_code1;
    char   sqlca_sqlstate  [5];
#define sqlstate sqlca_sqlstate
    char   sqlca_filler_2  [3];
#endif
    char   sqlca_wrn_area SQLCA_WARN_LEN&hyphen. ;
#define sqlca_warn0 sqlca_wrn_area[0]
#define sqlca_warn1 sqlca_wrn_area[1]
#define sqlca_warn2 sqlca_wrn_area[2]
#define sqlca_warn3 sqlca_wrn_area[3]
#define sqlca_warn4 sqlca_wrn_area[4]
#define sqlca_warn5 sqlca_wrn_area[5]
#define sqlca_warn6 sqlca_wrn_area[6]
#define sqlca_warn7 sqlca_wrn_area[7]
#define sqlwarn0    sqlca_wrn_area[0]
#define sqlwarn1    sqlca_wrn_area[1]
#define sqlwarn2    sqlca_wrn_area[2]
#define sqlwarn3    sqlca_wrn_area[3]
#define sqlwarn4    sqlca_wrn_area[4]
#define sqlwarn5    sqlca_wrn_area[5]
#define sqlwarn6    sqlca_wrn_area[6]
#define sqlwarn7    sqlca_wrn_area[7]
#ifdef DB2
    char   sqlcext       [5];
}
sqlca = {"SQLCA"  ",136,0,"
        "        ",0,0,0,0,0,0,0,"      ",0,0,0,0,0,0
};
#else
    char   sqlca_pgm_name [8];
    char   sqlca_authid  [18]
    char   sqlca_plan_name [18];
}
sqlca = {"SQLCA***",196,"10", "0 ", "      ",0,0,
        "        ", "      ", "      ", "      ", "      ",0, " ", 0,
        ' ', ' ', ' ', ' ', "      ", "      ", "CA AuthID", "<pgmname>"
};
#endif

```

## SQL State Classes

The two-character SQLSTATE classes, listed in the following table, are the first (left-most) characters of the SQLSTATE. You can extract these classes from SQLSTATEs returned by CA Datacom and use the error-class-sensitive criteria thus obtained for error recovery purposes. For a table listing the SQLSTATEs in numerical order and the SQL return code(s) that equate to them, see [SQL States Table](#) (see page 308).

Be aware that some classes shown in the following table may not yet have had associated error conditions defined for them by CA Datacom. Also, we reserve the right to add or drop classes and to move existing error conditions between classes and subclasses (the right-most 3 bytes) at any time, thereby changing the SQLSTATE associated with an error condition. However, such changes are expected to only rarely occur.

<b>Class</b>	<b>Description</b>
00	Successful completion.
01	Warning (completion condition as opposed to exception condition).
02	No data (completion condition as opposed to exception condition).
03	SQL statement not yet complete.
07	Dynamic SQL error.
08	Connection exception.
09	Triggered action exception.
0A	Feature not supported.
0B	Invalid transaction initiation.
0D	Invalid target type specification.
0F	Locator exception.
0L	Invalid grantor.
0P	Invalid role specification.
OW	Prohibited stmt encountered during trigger exec.
21	Cardinality violation.
22	Data exception.
23	Integrity constraint violation.
24	Invalid cursor state.
25	Invalid transaction state.
26	Invalid SQL statement name.

<b>Class</b>	<b>Description</b>
27	Triggered data change violation.
28	Invalid authorization specification.
2B	Dependent privilege descriptors still exist.
2D	Invalid transaction termination.
2E	Invalid connection name.
2F	SQL routine exception.
30	Invalid SQL statement.
31	Invalid target specification value.
33	Invalid SQL descriptor name.
34	Invalid cursor name.
35	Invalid condition number.
36	Cursor sensitivity exception.
38	External routine exception.
39	External routine invocation exception.
3B	Savepoint exception.
3C	Ambiguous cursor name.
3D	Invalid catalog name.
3F	Invalid schema name.
40	Transaction rollback.
42	Syntax error or access rule violation.
44	With check option violation.
51	Application state is invalid.
53	Inconsistent specification or invalid operand.
54	SQL or DATACOM limit exceeded.
55	Object not in prerequisite state.
56	Miscellaneous SQL or DATACOM error.
57	Resource unavailable or operator intervened.
58	System error.
80	CA Datacom Datadictionary, DDD, and CXX errors.
Hx	WHERE x=1 thru F (H1 thru HF): SQL Multimedia (various subclasses).

Class	Description
HZ	Remote Database Access (various subclasses).
Re	SQLCODE -118, where the e is the first character of the CA Datacom/DB return code.
Se	SQLCODE -117, where the e is the first character of the CA Datacom/DB return code.

## SQL States Table

This table lists the SQLSTATES in numerical order and the SQL return code(s) that equate to them. For information about the defined classes of SQLSTATES, see [SQL State Classes](#) (see page 306).

SQL State	SQL Return Code(s)
00000	0, -985, -986, -987, -988, -989, -990, -991, -992, -993, -994, -995, -996, -997, -998
01S01	+170
01S02	-243
07003	-134
07S01	-301
07S02	-302
07S03	-303
07S04	-304
07S05	-300
07S06	-306
07S07	-307
07S08	-308
07S09	-309
07S10	-311
07S11	-312
09S01	-533
09S02	-532
0AS01	-023, -024, -027, -029, -031, -109,

---

<b>SQL State</b>	<b>SQL Return Code(s)</b>
0AS02	-087
0AS03	-101
0AS04	-241
0AS05	-285
0AS06	-299
0AS07	-269
02000	+100
21000	-142, -148
2200C	-042
22001	-053
22002	-001
22003	-004
22007	-193, -198, -199, -200, -201, -202, -203, -204, -218, -219, -223, -224, -225, -226, -227, -228, -229, -230, -231, -232, -233, -234, -235, -236, -237, -238
22008	-193, -198, -199, -200, -201, -202, -203, -204, -230, -237
22504	-294, -296, -297, -298
22512	-293
22S01	-039
22S02	-295
22S03	-161
23502	-086
23503	-176
23504	-175
23505	-263
23513	-167
23515	-267
24S01	-180
24S02	-181
24S03	-183
24S04	-317

---

<b>SQL State</b>	<b>SQL Return Code(s)</b>
24S05	-130
24501	-135
24502	-502
25S01	-147
25004	-144
25501	-122
34S01	-153
34000	-133
38S01	-530
38S02	-531
38S04	-534
39S01	-535
39001	-321, -322
3FS01	-146
3FS02	-150
42S01	-025
42S02	-244
42S03	-245
42S04	-246
42S05	-270
42S06	-273
42S07	-313
42S08	-314
42S09	-315
42S10	-172
42S11	-563
42S12	-316
42S13	-075
42S14	-209
42501	-185

---

<b>SQL State</b>	<b>SQL Return Code(s)</b>
42502	-003, -008, -247, -248
42601	-020, -110
42603	-007
42604	-105, -284
42606	-565, -566
42607	-072
42611	-151
42612	-002
42622	-030
42625	-208
42701	-055, -069
42702	-033
42703	-009
42703	-261
42704	-014, -015, -034, -127, -179, -250
42707	-070
42710	-165
42802	-054, -057
42803	-073, -076, -103
42804	-207
42805	-066
42806	-159
42807	-079, -111
42808	-113
42811	-155
42815	-044, -278
42816	-277
42818	-041, -194, -195
42820	-289
42821	-058

---

<b>SQL State</b>	<b>SQL Return Code(s)</b>
42823	-071
42824	-290
42825	-083
42826	-082
42830	-173
42831	-182
42837	-255
42889	-168, -564
42890	-169
42902	-162, -174
42903	-067
42905	-080
42908	-184
44S01	-178
44S02	-018
44000	-156
51S01	-129
51S02	-210
51002	-124
51003	-120
53S01	-017
53S02	-019
53S03	-036
53S04	-043
53S05	-045
53S06	-084
53S07	-085
53S08	-132
53S09	-138
53S10	-139



---

<b>SQL State</b>	<b>SQL Return Code(s)</b>
53S11	-140
53S12	-141
53S13	-145
53S14	-143
53S15	-149
53S16	-163
53S17	-189
53S18	-190
53S19	-191
53S20	-192
53S21	-205
53S22	-206
53S23	-242
53S24	-254
53S25	-257
53S26	-260
53S27	-274
53S28	-286
53S29	-310
53S30	-503
53S31	-504
53S32	-505
53S33	-136
53S34	-275
53S35	-276
53S36	-305
53S37	-320
53S38	-562
53S39	-292
53S40	-215

---

<b>SQL State</b>	<b>SQL Return Code(s)</b>
53S41	-214
53S42	-216
54S01	-021
54S02	-164
54S03	-256
54S04	-319
54001	-032
54002	-006, -012
54004	-026, -152
54010	-061
54010	-131
54011	-060
55S01	-166, -249, -262
55S02	-252
55S03	-251
55S04	-037
55S05	-158
55S06	-171
55S07	-177
55S08	-188
55S09	-291
55S10	-318
56S01	-056
56S02	-123
56S04	-264
56S05	-038
56S06	-119
56S07	-121
56S08	-128
56S09	-265

---

<b>SQL State</b>	<b>SQL Return Code(s)</b>
56S10	-266
56S11	-272
56S12	-279
56S13	-280
56S14	-281
56S15	-282
56S16	-283
56S17	-287
56S18	-288
56SNO	-064
56S19	-999
57S01	-040
57S02	-125
57S03	-137
57S04	-259
57S05	-560, -561
57S06	-258
57S07	-016
57002	-559
57011	-010, -157
58S01	-126
58S02	-035
80S01	-005
REEII	-118
SEEII	-117

---



# Chapter 8: Application Tasks Using Embedded SQL

---

The following chapters contain examples on embedding SQL statements in a host program. The following table lists the chapters in this part and what each contains:

Chapter/Section	Contains
<a href="#">Specifying Result Tables</a> (see page 321)	Examples on different ways to use the select-statement to retrieve the desired data.
<a href="#">Selecting All Columns</a> (see page 321)	An example on selecting all columns in a table.
<a href="#">Selecting Some Columns</a> (see page 322)	An example on selecting specific columns in a table.
<a href="#">Selecting Using Search Conditions</a> (see page 323)	An example on using search conditions to limit the rows in the result table.
<a href="#">Ordering by Column Values</a> (see page 324)	An example on ordering returned data by values in the column(s).
<a href="#">Eliminating Duplicate Rows</a> (see page 325)	An example on eliminating duplicate rows in the returned data.
<a href="#">Counting</a> (see page 326)	An example on counting the rows in a result table.
<a href="#">Calculating Values</a> (see page 327)	Examples on using column values to calculate values which are not contained in the table, and also using a subquery (nested subselects).
<a href="#">Summarizing Group Values</a> (see page 329)	An example on grouping columns by value.
<a href="#">Testing for Existence</a> (see page 331)	An example on testing for existence of certain rows.
<a href="#">Selecting Data from Multiple Tables</a> (see page 333)	Discusses the join and union operations on data from multiple tables.
Joining Tables	An example on deriving a result table which includes all specified data from two or more tables.
<a href="#">Using the UNION Operator</a> (see page 337)	An example on deriving a result table which is a set formed by the union of two result tables.

Chapter/Section	Contains
<a href="#">Inserting Rows</a> (see page 341)	Examples on using the INSERT statement to insert rows into a table.
<a href="#">Updating a Table</a> (see page 343)	An example on using the UPDATE statement to update rows in a table.
<a href="#">Deleting Rows</a> (see page 347)	An example on using the DELETE statement to delete rows from a table.
<a href="#">Committing and Backing Out Transactions</a> (see page 349)	An example on using the COMMIT WORK statement to commit transactions and the ROLLBACK WORK statement to back out transactions.

The embedded examples include the following statements, clauses, functions and predicates.

Statements:	Clauses:	Functions:	Predicates:
CLOSE	FROM	AVG	EXISTS
COMMIT WORK	GROUP BY	COUNT	
DECLARE	HAVING	(DISTINCT	
CURSOR	ORDER BY	column-name)	
DELETE	WHERE	COUNT (*)	
FETCH		MAX	
INSERT		MIN	
OPEN		SUM	
ROLLBACK			
WORK			
SELECT			
SELECT INTO			
UPDATE			
WHENEVER			

With regard to the SELECT statement listed previously, note that in the examples the select-statement form of the SELECT statement is a component of the DECLARE CURSOR statement, and that the examples could be executed interactively by omitting the "DECLARE CURSOR FOR" clause. See DECLARE CURSOR for more information on the DECLARE CURSOR statement and [SELECT](#) (see page 766) for more information on the SELECT statement.

**Note:** All examples use the CUSTOMERS and ORDERS tables listed in [Sample Data Tables](#) (see page 823).

For examples using other SQL statements, such as CREATE and DROP, see the appropriate chapter in the section starting on [Using the Interactive SQL Service Facility](#) (see page 355).

For examples involving the use of dynamic SQL, see [Dynamic SQL](#) (see page 55). Also see the descriptions and examples of the dynamic SQL statements in [SQL Statements](#) (see page 597).





# Chapter 9: Specifying Result Tables

---

You can use the SELECT statement to construct queries which specify the result table you want.

The various clauses of the SELECT statement allow you to limit the data you retrieve to only that data which is significant for your purpose.

**Note:** In the following examples, the SELECT statement is embedded inside a DECLARE CURSOR statement, as it would be in a COBOL program.

## Selecting All Columns

You do not have to name every column in a table if you want to retrieve each one, as the following example shows.

### Problem

Select all columns from the table ORDERS.

### Solution

```
.  
. (COBOL statements)  
. .  
1 EXEC SQL  
2     DECLARE ORDER_LIST CURSOR FOR  
3     SELECT *  
4     FROM ORDERS  
5 END-EXEC  
. .  
. (COBOL statements)  
. .
```

### Line 3

The asterisk (\*) in the select-statement is used to retrieve all columns from the specified table without having to name each column.

## Selecting Some Columns

A result table can contain only those columns you specifically want to see. To retrieve specific columns, you name those columns in the SELECT statement, as shown in the following:

### Problem

Select some specific columns from table CUSTOMERS.

### Solution

```
      .  
      .  
      (COBOL statements)  
      .  
      .  
1 EXEC SQL  
2     DECLARE CUSTLIST CURSOR FOR  
3     SELECT CUST_NO, CITY, STATE  
4     FROM CUSTOMERS  
5 END-EXEC.  
      .  
      .  
      (COBOL statements)  
      .  
      .
```

### Line 3

To retrieve only specific columns for inclusion in your result table, name those columns in the SELECT statement. This example retrieves columns named CUST\_NO, CITY and STATE from the CUSTOMERS table to build a result table.

## Selecting Using Search Conditions

A search condition (WHERE clause) in the SELECT statement can limit the retrieved data to only that which is significant for your purposes.

### Problem

Select all columns from table CUSTOMERS, but only retrieve rows where the value of the column STATE is equal to the value of the host variable WS-STATE.

### Solution

```
      .  
      .  
      (COBOL statements)  
      .  
      .  
1 EXEC SQL  
2     DECLARE CUSTLIST CURSOR FOR  
3     SELECT *  
4     FROM CUSTOMERS  
5     WHERE STATE = :WS-STATE  
6 END-EXEC.  
      .  
      .  
      (COBOL statements)  
      .  
      .
```

### Line 3

All columns are selected for inclusion in the result table.

### Line 5

The search condition, STATE = :WS-STATE, limits the number of rows retrieved. Only rows where the column STATE contains a value equal to the value of the host variable WS-STATE are selected. Hyphens can be used in a COBOL item which is referenced in an SQL statement.

## Ordering by Column Values

When the ordering of retrieved data is important, use the ORDER BY clause.

### Problem

Specify columns CUST\_NO, CITY, and STATE for retrieval from the CUSTOMERS table. Place rows of the result table in ascending order by CUST\_NO.

### Solution

```
      .  
      .  
      (COBOL statements)  
      .  
      .  
1 EXEC SQL  
2     DECLARE CUSTLIST CURSOR FOR  
3     SELECT CUST_NO, CITY, STATE  
4     FROM CUSTOMERS  
5     ORDER BY CUST_NO  
6 END-EXEC.  
      .  
      .  
      (COBOL statements)  
      .  
      .
```

### Line 3

Three columns are selected for inclusion in the result table.

### Line 5

The ORDER BY clause specifies the order the rows are to be placed in the result table. The default order is ascending, so the rows are in ascending order, that is to say, the lowest customer identification number to the highest.

## Eliminating Duplicate Rows

Redundant duplicates can be eliminated from your result table by specifying the keyword `DISTINCT` in the `SELECT` statement.

### Problem

Retrieve all customer states in `CUSTOMERS`, eliminating any duplicates.

### Solution

```
      .  
      .  
      (COBOL statements)  
      .  
      .  
1 EXEC SQL  
2     DECLARE CUSTLIST CURSOR FOR  
3     SELECT DISTINCT STATE  
4     FROM CUSTOMERS  
5 END-EXEC.  
      .  
      .  
      (COBOL statements)  
      .  
      .
```

**Line 3** The keyword `DISTINCT` specifies that any redundant duplicate values for `STATE` are to be eliminated from the result table. The result table includes one occurrence of each unique value for `STATE`.

## Counting

The COUNT(\*) function lets you tally the number of rows in the result table. The COUNT(DISTINCT column-name) form of this function returns the number of distinct values in the specified column.

The following example uses the COUNT(DISTINCT column-name) function in a SELECT INTO statement.

### Problem

Find the number of unique customer states contained in CUSTOMERS.

### Solution

```
      .  
      .  
      (COBOL statements)  
      .  
      .  
1  EXEC SQL  
2      SELECT COUNT(DISTINCT STATE)  
3      INTO :WC-COUNT  
4      FROM CUSTOMERS  
5  END-EXEC.  
      .  
      .  
      (COBOL statements)  
      .  
      .
```

### Line 2

The COUNT(DISTINCT STATE) function tallies the number of distinct values in the STATE column of CUSTOMERS. The result of the SELECT statement is the result of the COUNT function, which is a number, not a result table.

### Line 3

The INTO clause specifies that the value returned by the COUNT function is placed in the host variable :WC-COUNT. Hyphens can be used in a COBOL item which is referenced in an SQL statement.

## Calculating Values

You can use expressions and/or functions in the SELECT statement to calculate values which are not contained in the actual table. Predicates which contain expressions can be used in the WHERE clause to form a search condition based on existing column values.

### Example 1

#### Problem

Calculate each year-to-date sales after a 10 percent discount.

#### Solution

```
      .  
      .  
      (COBOL statements)  
      .  
      .  
1 EXEC SQL  
2     DECLARE YTDLIST CURSOR FOR  
3     SELECT YTD_SALES, YTD_SALES * .9  
4     FROM CUSTOMERS  
5 END-EXEC.  
      .  
      .  
      (COBOL statements)  
      .  
      .
```

#### Line 3

The SELECT statement includes the expression `YTD_SALES * .9` to calculate the new year-to-date sales after a 10 percent discount.

## Example 2

The following example uses nested subselects. The inner subselect is called a *subquery*. Correlation names are also used in this example to avoid ambiguity in referring to columns.

### Problem

Find all year-to-date sales whose current value is greater than the average year-to-date sales, and the industry code is A.

### Solution

```
      .  
      .  
      (COBOL statements)  
      .  
      .  
1  EXEC SQL  
2      DECLARE YTDLIST CURSOR FOR  
3          SELECT CUST_NO, YTD_SALES  
4          FROM CUSTOMERS C1  
5          WHERE IND_CD = 'A'  
6              AND YTD_SALES >  
7                  (SELECT AVG(YTD_SALES)  
8                     FROM CUSTOMERS  
9                     WHERE CUST_NO = C1.CUST_NO)  
10 END-EXEC.  
      .  
      .  
      (COBOL statements)  
      .  
      .
```

### Lines 3-6

The outer subselect requests a set of all rows where the IND\_CD values is equal to 'A' and the YTD\_SALES value is greater than the specifications in the subquery. In Line 4, C1 is the correlation name for the CUSTOMERS table in the outer subselect.

### Lines 7-9

The subquery finds the average YTD\_SALES where the CUST\_NO value equals the CUST\_NO value in the outer subselect. C1 is used as a qualifier to indicate the reference is to CUST\_NO in the outer subselect.



## Summarizing Group Values

Use the GROUP BY clause to apply a function to each group of column values. Except for the group column(s), any other column you specify can be the argument of a column function.

### Problem

Show the state, maximum year-to-date sales, minimum year-to-date sales, and average year-to-date sales for each state in the CUSTOMERS table. Each group must have more than one row and the maximum year-to-date sales must be more than \$200,000.

### Solution

```
      .  
      .  
      (COBOL statements)  
      .  
      .  
1 EXEC SQL  
2     DECLARE YTDLIST CURSOR FOR  
3         SELECT STATE, MAX(YTD_SALES), MIN(YTD_SALES), AVG(YTD_SALES)  
4         FROM CUSTOMERS  
5         GROUP BY STATE  
6         HAVING COUNT(*) > 1 AND MAX(YTD_SALES) > 200000  
7 END-EXEC.  
      .  
      .  
      (COBOL statements)  
      .  
      .
```

**Line 3**

The SELECT statement finds the maximum and minimum year-to-date sales and calculates the average year-to-date sales.

**Line 5**

The GROUP BY clause specifies that each function in the SELECT statement is to be applied to each group of a STATE value and one row is to be returned for each distinct state.

**Line 6**

The HAVING clause limits the result table to only those groups where the STATE value was found more than once and the maximum year-to-date sales for the state was greater than \$200,000.

## Testing for Existence

Use the EXISTS predicate to test for the existence of certain rows. The EXISTS predicate evaluates to *true* only if the subquery finds a row which meets the specifications of its search condition.

### Problem

Select the state and minimum year-to-date sales for each state, but only check customers who have orders.

### Solution

```
      .  
      .  
      (COBOL statements)  
      .  
      .  
1 EXEC SQL  
2     DECLARE YTDLIST CURSOR FOR  
3     SELECT STATE, MIN(YTD_SALES)  
4     FROM CUSTOMERS  
5     WHERE EXISTS  
6     (SELECT *  
7     FROM ORDERS  
8     WHERE ORDERS.CUST_NO = CUSTOMERS.CUST_NO)  
9     GROUP BY STATE  
10 END-EXEC.  
      .  
      .  
      (COBOL statements)  
      .  
      .
```

**Line 3**

The outer subselect of the SELECT statement specifies what columns to retrieve only if the subselect of the EXISTS predicate (lines 5-8) evaluates to *true*.

**Lines 5-8**

The subselect of the EXISTS predicate specifies which conditions must be met in order for the predicate to evaluate to *true*. In this case, the customer number in the ORDERS table must match a customer number in the CUSTOMERS table.

**Line 9**

The columns of the result table formed by the outer subselect are grouped by the value of the STATE column, that is to say, the result table contains one row for each unique value of STATE.

# Chapter 10: Selecting Data from Multiple Tables

---

You can specify result tables which select data from two or more base tables, two or more views, or even other result tables.

One method is the **join**, which derives a result table that includes the specified columns from two or more base tables, views, or other result tables.

Another method is the **union**, which produces a result table that is actually a set derived from two or more result tables. This method, using the UNION operator, eliminates duplicate rows in the result table. To retain duplicate rows in the set, specify UNION ALL.

## Joining Tables

The ability to join two or more tables (and/or views) easily is a major advantage that distinguishes relational systems from non-relational systems. The join capability simplifies the task of retrieving data from different tables to build a single result table holding all the necessary data.

You implement this join by forming a query which retrieves data from more than one table. Your SELECT statement includes columns (qualified by table name) from two or more tables. The FROM clause of your query names the tables used as qualifiers in the select-statement.

All the columns specified in the SELECT form the result table. Therefore, if you specify the column CUST\_NO from the CUSTOMERS table and the column CUST\_NO from the ORDERS table, your result table includes two CUST\_NO columns, each qualified by the original table name.

You can reference up to 20 tables in a FROM clause when you are performing a join. For example, if a view is based on five tables, you can name that view in the FROM clause, and up to fifteen other tables.

There is a special case where one or more tables are not needed and can be eliminated from a query, so that the query executes more efficiently. This case usually occurs when a view is defined that includes optional tables, but the particular use of the view does not require all the tables.

Because it is an optional table, it is accessed using a LEFT JOIN, which does not change the number of rows returned from the primary table on the left side of the join when there is no matching row. When the join is on a unique/primary key, the most rows that can be found is one, and therefore the count of rows is also not changed when the single matching row is found. Finally, if no columns in the optional table are referenced in the query (other than the left join condition, of course), this optional table has been accessed and not used. It is, therefore, eliminated from the query for the purpose of better performance.

Example:

```
CREATE VIEW INSURANCE AS
SELECT * FROM ACCOUNT
T1 LEFT JOIN CARINS T2 ON T1.COL1 = T2.COL1
T1 LEFT JOIN HOMEINS T3 ON T1.COL1 = T3.COL1
T1 LEFT JOIN LIFEINS T4 ON T1.COL1 = T4.COL1 ;

SELECT T1.NAME, T2.LICENSE FROM INSURANCE WHERE T2.VIN = :HOSTVAR ;
```

In this example, where COL1 is the Primary key in all tables, only the ACCOUNT and CARINS tables are referenced. It doesn't matter if there is a single matching HOMEINS or LIFEINS row found, and therefore these tables are eliminated from the query.

Also see [Left Outer Joins](#) (see page 110).

### Example

The following example joins the CUSTOMERS and ORDERS tables. This example is taken from DBCOBSQA, a sample program available on the installation tape.

**Note:** You can only join tables which have the same security type, that is to say, either the CA Datacom/DB External Security Model or the SQL Security Model. See the *CA Datacom Security Reference Guide* for more information about Security Models.

#### Problem

List the customer number, name and order ID for those customers who have outstanding orders.

#### Solution

```
.  
.  
  (COBOL statements)  
.  
.  
1 EXEC SQL  
2   DECLARE CUSTORD CURSOR FOR  
3     SELECT CUSTOMERS.CUST_NO, ORD_ID, NAME  
4       FROM CUSTOMERS, ORDERS  
5        WHERE CUSTOMERS.CUST_NO = ORDERS.CUST_NO  
6        ORDER BY CUSTOMERS.CUST_NO  
7 END-EXEC  
.  
.  
  (COBOL statements)  
.  
.
```

#### Line 3

The SELECT statement specifies the columns to be selected from each table. Column names which are the same in each table are qualified by the table name, such as CUSTOMERS.CUST\_NO. ORD\_ID is not qualified since it exists only in the ORDERS table and NAME is not qualified since it exists only in the CUSTOMERS table.

#### Line 4

Both tables are named in the FROM clause, indicating that the result table includes the retrieved data from each table.

**Line 5**

The WHERE clause specifies that the value of the CUST\_NO column in each table must be equal to be selected for the result table. If a customer does not have an outstanding order, then the CUST\_NO value does not appear in the ORDERS, nor in the result of the join. The comparison is possible since the columns have comparable data types. For comparison rules, see Basic Operations (Assignment and Comparison).

**Line 6**

The ORDER BY clause specifies that the rows in the result table be in ascending order according to the customer number.

**Sample Output**

Following is the report produced by running DBCOBSQA.

----- CURRENT ORDERS -----PAGE 1	
CUSTOMER NO	CUSTOMER NAME
-----	-----
0030	CANNON TOOLS CO
0230	CHEMICAL MUTUAL
1210	LINGBERGH INDUSTRIES
1210	LINGBERGH INDUSTRIES
1450	UNION TRANSPORTATION
1630	MARBURY MATERIALS
1850	TECH CASTLE RESEARCH
1890	FIRST STREET BANK CORP
2050	TRANSAMERICAN PUBLISHING
END OF REPORT	



## Using the UNION Operator

Using the UNION operator derives a result table by combining two other result tables.

The set of rows in the UNION of result tables R1 and R2 is the set of rows in either R1 or R2, with redundant duplicate rows eliminated. Each row of the UNION table is either a row from R1 or a row from R2.

The columns of the result table are not named.

### **Duplicate Rows**

Two rows are duplicates of one another only if each value in the first row is equal to the corresponding value of the second row.

All but one row of each set of duplicates are eliminated by a UNION. The number of rows in the UNION table is the sum of the number of rows in R1 and R2, less the number of duplicates eliminated.

If you specify UNION ALL, duplicate rows are not eliminated.

### Rules for Columns

Result tables R1 and R2 must have the same number of columns.

With the exception of column names, the description of the first column of R1 must be identical to the description of the first column of R2, that is to say, the data type and the length must be the same. The description of the second column of R1 must be identical to the description of the second column of R2, and so on.

### Example

The following example (see next page) performs a union on the result tables derived from the CUSTOMERS and ORDERS tables. This example is taken from DBCOBSQF, a sample program available on the installation tape.

### Problem

List all customer numbers for customers who have more than \$300,000 in year-to-date sales.

### Solution

```
      .  
      .  
      (COBOL statements)  
      .  
      .  
1 EXEC SQL  
2   DECLARE CUSTORD2 CURSOR FOR  
3     SELECT CUST_NO  
4       FROM CUSTOMERS  
5       WHERE YTD_SALES > 300000  
6     UNION  
7     SELECT CUST_NO  
8       FROM ORDERS  
9     ORDER BY 1  
10  END-EXEC  
      .  
      .  
      (COBOL statements)  
      .  
      .
```

**Lines 3-5**

The first subselect specifies the column to retrieve from the CUSTOMERS table. The search condition in the WHERE clause limits the retrieved rows to those where the value of YTD\_SALES is greater than \$300,000. You do not have to qualify the CUST\_NO column in this SELECT to distinguish it from the CUST\_NO column of the ORDERS table because each subselect is evaluated separately. The union is performed after each subselect has been processed.

**Line 6**

The UNION operator between the two subselect statements means the final result table contains data that is a set formed from the data retrieved by each subselect.

**Lines 7-9**

The second subselect specifies the column to retrieve from the ORDERS table. No search condition limits the number of rows retrieved by this subselect. The ORDER BY clause specifies that the rows in the result table of this union are to be placed in ascending order according to the value in the first column. This column is referenced by a number because columns do not have names in a result table formed by a union operation.

In the previous example, each subselect has one column and the definition of the column in the first subselect is identical to the definition of the column in the second subselect.

The union of the two result tables would not be possible if one subselect had more columns specified than the other, or the column definitions did not match.

The result table formed by the UNION operation is a set of the data retrieved by each subselect. The columns of the result table are not named.

### Sample Output

Following is the report produced by running DBCOBSQF:

```
----- CURRENT ORDERS -----PAGE 1
CUSTOMER NO
-----
0030
0170
0230
1210
1450
1630
1850
1890
1950
1970
2010
2050
2070
2090
2250
2330
2690
3910
4310
4350
5590
6390
7150
7290
7350
7410
7790
9130
END OF REPORT
```

# Chapter 11: Inserting Rows

---

To insert rows into a table or view, use the INSERT statement. The values you assign to columns during the insert can be literal values or values which have been placed in host variables.

Literal values assigned to columns that contain character data must be enclosed in apostrophes ('). Literal values assigned to columns of numeric data types are not enclosed in apostrophes.

The data type of the host variable must be compatible with the data type of the column to which the value is being assigned.

If you specify the column names in the INSERT statement, the values for the columns must be listed in the same order as the column names. This is shown in Example 1.

If you do not specify the column names in the INSERT statement, the values for the columns must be listed in the same order as the columns are specified in the base table (see Example 2).

## Example 1

### Problem

Add rows to the CUSTOMERS table where the value of CUST\_NO and NAME are assigned from the host variables WC-CUSTNO and WC-NAME.

### Solution

```
      .  
      .  
      (COBOL statements)  
      .  
      .  
      (loop until end of input)  
      .  
      .  
1 EXEC SQL  
2     INSERT INTO CUSTOMERS (CUST_NO, NAME)  
3     VALUES (:WC-CUSTNO, :WC-NAME)  
4 END-EXEC.
```

### Lines 2-3

The row is inserted in the CUSTOMERS table with the values in the host variables assigned to the columns CUST\_NO and NAME.

## Example 2

The following problem shows how to insert a row in a table without having to specify column names to assign a value to each column.

### Problem

Add a row to the CUSTOMERS table with values specified for each column. The values must be specified in the same order as the columns are listed in the table definition.

### Solution

```
      .  
      .  
      (COBOL statements)  
      .  
      .  
1 EXEC SQL  
2     INSERT INTO CUSTOMERS  
3     VALUES ('Z',  
4             '9999',  
5             'LAGOONS R US',  
6             '925 DAILY DRIVE',  
7             'BOX 25',  
8             'AGANA',  
9             'GU',  
10            '89333',  
11            'B',  
12            '808',  
13            '967',  
14            '2774')  
15 END-EXEC.
```

### Lines 2-14

The row is inserted into the CUSTOMERS table with the specified value for each column. Since the values are listed in the same order that the columns are specified in the table definition, it is not necessary to name the columns.

# Chapter 12: Updating a Table

---

Use an UPDATE statement to modify the contents of one or more rows. All rows in a table that satisfy the search condition are updated in accordance with the assignments in the SET clause. You can update only one table in a single statement. Following is an example:

## Problem

Initialize new records (those with STATE equal to WS-STATE) with year-to-date sales equal to the host variable WS-YTD-SALES, and the salesman ID equal to the host variable WS-SLMN-ID.

## Solution

```
.
.
(COBOL statements)
.
1 EXEC SQL
2     DECLARE CUSTUPD CURSOR FOR
3     SELECT *
4     FROM CUSTOMERS
5     WHERE STATE = :WS-STATE
6 END-EXEC.

7 EXEC SQL WHENEVER NOT FOUND GOTO PROGEND END-EXEC.
8 EXEC SQL WHENEVER SQLWARNING CONTINUE END-EXEC.
9 EXEC SQL WHENEVER SQLERROR GOTO ERRORTN END-EXEC.
.
.
10 EXEC SQL OPEN CUSTUPD END-EXEC.
11 PERFORM PROCESS-CUSTOMERS-LOOP THROUGH
12     END-PROCESS-CUSTOMERS-LOOP.
13 EXEC SQL CLOSE CUSTUPD END-EXEC.
.
.
```

```
14  PROCESS-CUSTOMERS-LOOP.  
15      EXEC SQL  
16          FETCH CUSTUPD INTO  
17              :WC-IND-CD,  
18              :WC-CUSTNO,  
19              :WC-NAME,  
20              :WC-ADDR-1,  
21              :WC-ADDR-2,  
22              :WC-CITY,  
23              :WC-STATE,  
24              :WC-ZIP,  
25              :WC-CRED-IND,  
26              :WC-AREA-CD,  
27              :WC-PH-EXCH,  
28              :WC-PH-NO,  
29              :WC-OPEN-DOL,  
30              :WC-YTD-SALES,  
31              :WC-ACT-YR,  
32              :WC-ACT-MO,  
33              :WC-ACT-DAY,  
34              :WC-SLMN-ID  
35      END-EXEC.  
36      IF SQLCODE = ZERO  
37          EXEC SQL  
38              UPDATE CUSTOMERS  
39                  SET YTD_SALES = :WS-YTD-SALES,  
40                      SLMN_ID = :WS-SLMN-ID  
41                  WHERE CURRENT OF CUSTUPD  
42      END-EXEC.  
.  
.  
43  END-PROCESS-CUSTOMERS-LOOP.
```

**Line 5**

The search condition in the SELECT of the DECLARE CURSOR statement specifies that the temporary result table contains rows only where the value of the STATE column is equal to the value of the host variable WS-STATE.

**Line 7**

The exception condition, NOT FOUND, directs the program to go to a program end routine if an SQL return code of +100 is received.

**Line 8**

The exception condition, SQLWARNING, directs the program to continue execution if a warning condition or a positive SQL return code other than +100 is received.



**Line 9**

The exception condition, `SQLERROR`, directs the program to go to an error handling routine in the host program if a negative SQL return code is received.

**Line 10**

The cursor named in the `DECLARE CURSOR` statement is opened and positioned *before* the first row of the result table formed by the `SELECT` in the `DECLARE CURSOR` statement.

**Lines 11-12**

The processing loop is performed. This loop contains the `FETCH` statement to position the cursor.

**Line 13**

The cursor named in the `DECLARE CURSOR` statement is closed.

**Lines 16-34**

The `FETCH` statement positions the cursor on the first (or next) row of the temporary result table and places that row into the host variables listed in this statement.

**Lines 38-41**

The value of each named column is updated in the row where the cursor is currently positioned.



# Chapter 13: Deleting Rows

---

To delete rows from a table or view use the DELETE statement. Deleting a row from a view deletes the row from the table that contains the row.

You can use the searched form of the DELETE statement or the positioned form with a cursor.

The searched form of the DELETE statement uses a WHERE clause to specify a search condition. Any row which matches the search condition in the WHERE clause is deleted.

The positioned form of the DELETE statement uses the CURRENT OF clause to name a cursor which has previously been declared. Only the row where the cursor is positioned is deleted. Following is an example.

**Important!** The DELETE is a powerful statement, and can delete all rows of a table if you neglect to specify a WHERE clause that limits the deletion.

## Problem

Delete all rows from the CUSTOMERS table where the STATE column contains the value specified in the host variable WS-STATE.

## Solution

```
.  
. (COBOL statements)  
. .  
1 EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.  
2 EXEC SQL WHENEVER SQLWARNING CONTINUE END-EXEC.  
3 EXEC SQL WHENEVER SQLERROR GO TO ERRORRN END-EXEC.  
4 MOVE 'GU' TO WS-STATE.  
5 EXEC SQL  
6     DELETE  
7     FROM CUSTOMERS  
8     WHERE STATE = :WS-STATE  
9 END-EXEC.
```

**Line 1**

The exception condition, NOT FOUND, directs the program to continue execution if an SQL return code of +100 is received.

**Line 2**

The exception condition, SQLWARNING, directs the program to continue execution if a warning condition or a positive SQL return code other than +100 is received.

**Line 3**

The exception condition, SQLERROR, directs the program to go to an error handling routine in the host program if a negative SQL return code is received.

**Lines 6-8**

The DELETE statement deletes all rows where the value of the STATE column equals the value of the host variable WS-STATE.

# Chapter 14: Committing and Backing Out Transactions

---

Use the COMMIT WORK statement to commit any changes made to the database when updating tables. Uncommitted changes to a database can be backed out with the ROLLBACK WORK statement. In the following example, inserts to the table are committed if no errors are detected during processing. If an error is detected, the inserts are backed out. Following is an example:

## Problem

Commit your transactions if no error is detected during processing. If an error is detected, rollback the transactions.

Solution

```
.
.
(COBOL statements)
.
.
1     IF SQLCODE = 0
2         MOVE 'INSERT COMPLETE' TO MSG-REC1
3         PERFORM COM-WORK
4         GO TO WRITE-REC.
5     IF SQLCODE > 100
6         MOVE 'WARNINGS ON INSERT' TO MSG-REC1
7         GO TO WRITE-REC.
8     IF SQLCODE < 0
9         PERFORM ROLBK-WORK
10        GO TO ERRORTN.
11    COM-WORK.
12        EXEC SQL
13            COMMIT WORK
14        END-EXEC.
15    ROLBK-WORK.
16        EXEC SQL
17            ROLLBACK WORK
18        END-EXEC.
```

The COBOL IF sets the condition for committing or backing out the inserts to the table. If the SQLCODE value in the SQL Communication Area (SQLCA) is 0 (zero), no errors were detected during processing. Any other SQLCODE value indicates an error has been detected.

**Lines 1-4, 11-14**

In this example, if no errors are detected, the inserts to the table are committed by the COMMIT WORK statement.

**Lines 5-7**

The program handles the case where a condition code greater than 100 is returned by issuing a message.

**Lines 8-10, 15-18**

If the SQLCODE value is less than 0, the ROLLBACK WORK statement is executed to back out the inserts.

# Chapter 15: Overview of the Interactive SQL Service Facility

---

The following chapters describe the Interactive SQL Service Facility of CA Datacom Datadictionary online. The Service Facility is designed primarily to allow you to create and test SQL objects and execute certain SQL statements. See the list of SQL statements that you can submit in [Executable SQL Statements](#) (see page 356).

- [Using the Interactive SQL Service Facility](#) (see page 355)

Overview of the Interactive SQL Service Facility and using this mode in CA Datacom Datadictionary online. The following sections contain detailed information.

- [Executable SQL Statements](#) (see page 356)
- [Specifying Unique SQL Names](#) (see page 356)
- [Submitting SQL Statements](#) (see page 357)
- [Using Commands](#) (see page 366)
- [COPY SQL Command](#) (see page 368)
- [DELETE SQL Command](#) (see page 370)
- [DISPLAY SQL Command](#) (see page 371)
- [EDIT SQL Command](#) (see page 372)
- [EXECUTE Command](#) (see page 373)
- [REBIND Command](#) (see page 373)
- [SCROLL Command](#) (see page 374)
- [Using Line Commands](#) (see page 375)
- [Using Margin Commands](#) (see page 376)
- [Using PF Keys](#) (see page 377)
- [Maintaining Source and Output Members](#) (see page 379)
- [Editing and Executing Source Members](#) (see page 380)
- [Displaying Source and Output Members](#) (see page 388)
- [Copying Source Members](#) (see page 393)
- [Deleting Source and Output Members](#) (see page 395)

- Creating SQL Objects

Overview of the Data Definition Language statements you submit through the Interactive SQL Service Facility to create objects or alter a table. The following sections contain step-by-step instructions and examples.

- Creating a Schema
- Creating a Table
- Altering a Table
- Creating an Index
- Creating a View



- Creating a Synonym
- Adding and Replacing Comments
- Deleting SQL Objects
  - Overview of the DROP statement and its impact on definitions in CA Datacom Datadictionary. The following sections contain step-by-step instructions and examples.
  - Deleting a Schema
  - Dropping an Index
  - Dropping a Table
  - Dropping a View
  - Dropping a Synonym
- Manipulating Data in SQL Tables
  - Overview of the Data Manipulation Language statements you can submit through the Interactive SQL Service Facility.
- Controlling Access Through SQL Statements
  - Overview of the Data Control Language statements you can submit through the Interactive SQL Service Facility.
- Performing SQL Administrative Tasks
  - Overview of the administrative tasks which can be performed for SQL processing using the Interactive SQL Service Facility. The following sections contain step-by-step instructions and examples.
  - Setting the Session Authorization ID
  - Deleting a Plan
  - Rebinding a Plan
  - Displaying Index of SQL Plans
  - Specifying Plan Options in a Source Member
  - Coding Plan Options

**Note:** For information about the CA Datacom/DB implementation of support for procedures and triggers, see [Procedures and Triggers](#) (see page 70) and [Datadictionary Support for Triggers and Procedures](#) (see page 86).



# Chapter 16: Using the Interactive SQL Service Facility

---

You can use the online panels in the CA Datacom Datadictionary Interactive SQL Service Facility for data definition, manipulation and control, and to perform administrative tasks associated with SQL usage. You must be authorized in CA Datacom Datadictionary to use the Interactive SQL Service Facility. See the person responsible for CA Datacom Datadictionary security on your system for authorization.

You should be familiar with the SQL relational sub-language and its languages, listed following, before you use the Interactive SQL Service Facility.

## **Data Definition Language (DDL)**

You can use specific SQL statements to create indexes, tables, and views, create schemas, create alternative names (synonyms) for tables and views, and enter descriptive information. You can also remove definitions.

**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)**

You can use specific SQL statements to insert, update, and delete the data in the production database.

## **SQL Control Statements**

includes the CALL and EXECUTE PROCEDURE statements that support the CA Datacom/DB implementation of procedures and triggers (see [Procedures and Triggers](#) (see page 70) and [Datadictionary Support for Triggers and Procedures](#) (see page 86)).

## Executable SQL Statements

You can execute the following SQL statements in the Interactive SQL Service Facility of CA Datacom Datadictionary online. The following table lists the SQL statements in each language:

Data Definition Language (DDL)	Data Manipulation Language (DML)	SQL Control Statements
ALTER TABLE	<b>Non-cursor operations:</b>	CALL
COMMENT ON	DELETE (searched DELETE)	EXECUTE PROCEDURE
CREATE INDEX	INSERT	
CREATE PROCEDURE	SELECT	
CREATE RULE	UPDATE (searched UPDATE)	
CREATE SCHEMA		
CREATE SYNONYM		
CREATE TABLE		
CREATE TRIGGER		
CREATE VIEW		
DROP		
GRANT		
REVOKE		

## Specifying Unique SQL Names

SQL names for schemas, tables, indexes, views, and synonyms must be unique according to the rules in the following table.

The SQL Statement	Defines In CA Datacom Datadictionary	SQL Name Requirements
CREATE SCHEMA	An AUTHORIZATION occurrence	The SQL name and the AUTHORIZATION occurrence name are the same and must be unique for all schemas.
CREATE TABLE	A TABLE occurrence	The SQL name of the table must be unique for all indexes, views and synonyms owned by a specific schema (authorization ID).

The SQL Statement	Defines In CA Datacom Datadictionary	SQL Name Requirements
CREATE INDEX	A KEY occurrence	The SQL name of the index must be unique for all indexes owned by a specific schema (authorization ID).
CREATE VIEW	A VIEW occurrence	The SQL name of the view must be unique for all tables, views and synonyms owned by a specific schema (authorization ID).
CREATE SYNONYM	A SYNONYM occurrence	The SQL name of the synonym must be unique for all tables, views and synonyms owned by a specific schema (authorization ID).

See the chapter on the SQL transport utility (DDTRSLM) in the *CA Datacom Datadictionary Batch Reference Guide* for information about additional restrictions on words used for an AUTHID, SQL name, or CA Datacom Datadictionary occurrence name.

When you create SQL tables, views and synonyms, the SQL name is prefixed by the authorization ID to create the TABLE, VIEW and SYNONYM occurrence name. The format is *authid-sqlname*.

Together, the authorization ID and SQL name of each table, view and synonym must be unique within the schema. For example, the names JONES.DEPTTBL (for a table) and JONES.DEPTTBL (for a view) **are not** unique since both are owned by the JONES schema, but the names JONES.DEPTTBL (for a table) and SMITH.DEPTTBL (for a view) **are** unique because they are owned by different schemas.

## Submitting SQL Statements

Before you can use CA Datacom Datadictionary online to access the Interactive SQL Service Facility, you must be authorized in CA Datacom Datadictionary to use this mode. See the person responsible for maintaining CA Datacom Datadictionary on your system for authorization.

## How to Submit SQL Statements

To submit SQL statements in the Interactive SQL Service Facility:

1. Place your SQL statements in the variable-line area of the Source Panel (examples are shown in the following sections). See Basic Language Elements for information on how to write SQL statements.
  - You can specify a name for the source member that is created when you execute your statements, or use a default source member.
  - You must place a semicolon after each complete SQL statement you enter.
  - You can use line commands to insert, delete, copy and move lines in the Source Panel.

The source member is saved by CA Datacom Datadictionary in a Virtual Library System (VLS) member. The name of the VLS library is specified in the System Resource Table parameter DDOLSQL= (in the DDSYSTBL macro). See the *CA Datacom/DB Database and System Administration Guide*.

You can optionally display, copy, modify, execute, and delete your source members through menu and panel selections or command line commands.

**Note:** You can put comment lines in your source member. Two dashes (--) indicate that the line is a comment. These two dashes *must* be in column one and two on the input line. Comment lines are included in the final tally of records within the member using the statement "number of records read is...", but they are not regarded as SQL statements. Blank lines can be used as separator lines. As with comment lines, blank lines are counted in the final record count but ignored as statements.

2. Submit the source member for processing with the EXECUTE command (PF9 key).
  - The Interactive SQL Service Facility prepares and executes the statements.
  - If successful, the CA Datacom Datadictionary is updated with the appropriate definitions in PROD status. Table objects are cataloged to the Directory and are ready to populate with data.
  - If multiple SQL commands are entered in a single source member and an SQL processing error (indicated by an SQL return code in the format *-nnn*) is encountered, a ROLLBACK WORK command is issued by the Interactive SQL Service Facility.

**Note:** A COMMIT WORK command is implied, in CICS, by a transaction boundary (that is to say, from EXECUTE to display of results is a single transaction). A ROLLBACK WORK is implied, in all environments, by a negative SQL return code (in the format *-nnn*) on any SQL statement in the member being executed. The COMMIT WORK and ROLLBACK WORK commands can be placed in a source member to be executed, if necessary.

3. CA Datacom Datadictionary places the results of processing the source member in an output member which is displayed on an Output Panel. The output member is saved in the same Virtual Library System (VLS) member as the source member.
  - The Output Panel indicates the status of the statement execution.
  - If the execution is not successful, CA Datacom Datadictionary displays return codes on this panel. Return codes for SQL and other related CA Datacom products are listed in the *CA Datacom/DB Message Reference Guide*.
  - You can display and delete the output members.

## How to Use

Use the following steps to submit your SQL statements.

### Step 1

When you sign on or select the SET MODE function, CA Datacom Datadictionary displays the Datadictionary Mode Select Panel. Select Option 7 or enter the SET MODE SQL command.

```

=>
=>
=>
-----
Datadictionary Mode Select
Enter desired option number ==> __ (There are 09 options on the menu) P00M
1. DBMAINT (SET MODE DBM) DATACOM/DB Structure Maintenance
2. ENTMAINT (SET MODE ENTM) Datadictionary Entity Maintenance
3. ENTDISPL (SET MODE ENTD) Datadictionary Entity Display
4. AUTHORIZE (SET MODE AUTH) Datadictionary Authorization Maintenance
5. FILEMAINT (SET MODE FMM) File Structure Maintenance
6. ISF (SET MODE ISF) Interactive Service Facility
7. SQL (SET MODE SQL) Interactive SQL Service Facility
8. IDEAL (IDEAL) Transfer to IDEAL application
9. OFF (OFF) End session

```

**Note:** If CA Ideal is not installed on your system, the OFF option appears in place of the CA Ideal option on the previous panel.

Also note that when you access the Interactive SQL Service Facility, CA Datacom Datadictionary checks for the existence of the relationship between your PERSON occurrence and a valid SQL AUTHORIZATION occurrence. If a relationship does not exist, CA Datacom Datadictionary presents the Default Authorization Panel. When the relationship exists, the Default Authorization Panel is not presented.

### Step 2

The Interactive SQL Service Facility Panel appears after you select this mode. Select Option 1 (SQLMAINT).

```
=>
=>
=>

-----
Interactive SQL Service Facility

Enter desired option number ==> __ (There are 05 options on the menu)

 1. SQLMAINT          SQL member maintenance/execution
 2. SQLADMIN         SQL administrative functions
 3. SET MODE         Reset Datadictionary processing mode
 4. IDEAL            Transfer to IDEAL application
 5. OFF              End session
```

**Note:** If CA Ideal is not installed on your system, the OFF option appears in place of the CA Ideal option on the previous panel.

### Step 3

CA Datacom Datadictionary displays the SQLMAINT menu panel. Select Option 1 or enter the EDIT SQL command.

```
=>
=>
=>

-----
Interactive SQL Service Facility                               SQLMAINT

Enter desired option number ==> __ (There are 05 options on the menu)

 1. EDIT              Edit/Execute SQL members
 2. DISPLAY           Display source/output members
 3. DELETE            Delete source/output members
 4. COPY              Copy source members
 5. END               End SQLMAINT processing
```



**Step 4**

CA Datacom Datadictionary displays the SQL Member Selection Criteria Fill-in Panel for the selected function. After reading the following list, complete your entries on the panel and press Enter:

```

=>
=>
=>
-----
Interactive SQL Service Facility                                SQLMAINT
                    SQL Member Selection Criteria Fill-in      S01F

EDIT   SQL / _____ , _____ , _____ /
EDT    (source) (output) (description)
        (name ) (name )

NOTE: If no source/output member is entered the default is $DDSQL.

PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD   PF8=FORWARD
PF9=EXECUTE   PF10=LEFT     PF11=RIGHT     PF12=ALTERNATE

```

**source name**

*(Optional)* Enter the name of the member that contains the SQL statements. Use alphanumeric characters in the name. It cannot contain special characters or embedded blanks.

- If the member does not exist, a new member is created.
- If the member already exists in the list of members created by the current user, the next panel displays the existing source member for modification.
- If you leave this field blank, the default name is used.

**Valid Entries:**

1- to 8-character member name

**Default Value:**

\$DDSQL

**output name**

*(Optional)* Enter the name of the member that contains the results of executing the source member. Use alphanumeric characters in the name. It cannot contain special characters or embedded blanks.

For convenience of identification, use the same name for both the source and output members.

- If the member already exists in the list of members created by the current user, CA Datacom Datadictionary replaces the previous output member upon execution.
- If you leave this field blank, the default name is used.

**Valid Entries:**

1- to 8-character member name

**Default Value:**

\$DDSQL

**description**

*(Optional)* You can enter up to 32 characters and embedded blanks to describe your source members. The same description is displayed with the output member.

**Note:** We recommend that you do not use a slash (/) as part of the description's text. The member cannot be retrieved with an EDT margin command if the description contains a slash.

**Valid Entries:**

1 to 32 characters

**Default Value:**

(No default)

**Step 5**

CA Datacom Datadictionary displays the Source Panel.

```

=>
=>
=>
-----
Interactive SQL Service Facility                               SQLMAINT
                                     Source Panel                S01S
      EDIT      Member: $DDSQL   Output Line Limit: 01000
                Person: JONES
      Current Authid: JONES
      Description:
-----
===== T O P =====
000001
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT    PF12=ALTERNATE

```

**Member:**

The source member name you entered on the SQL Member Selection Criteria Fill-in Panel or the default member name \$DDSQL.

**Output Line Limit**

The maximum number of lines you receive on the output panel. You can move the cursor to this field and change this limit. If your expected output exceeds 1000 lines, you should consider embedding your SQL statement(s) in a host program.

**Person:**

The PERSON occurrence name associated with the user ID entered on the CA Datacom Datadictionary Sign-on Panel.

**Current Authid:**

One of the following is displayed:

- The authorization ID established for the session with the SET AUTHID function issued with the command or the function in the SQL Administrative option.
- The default authorization ID related to the user ID.

**Description:**

The description you entered on the SQL Member Selection Criteria Fill-in Panel.

**numbered line(s)**

Use this area for your SQL statements.

- If you select an existing source member, the contents of the source member are displayed for you to modify or execute.
- If this is a new source member, you receive one numbered blank line. If the statement can be entered on one line, it is not necessary to insert additional lines on the panel.
- You can use line commands such as I (insert), R (repeat), or D (delete), to add and delete lines. See [Using Line Commands](#) (see page 375) for more information.
- CA Datacom Datadictionary allows you to enter your statements in mixed-case letters. If you enter mixed-case letters on a Source Panel, the Output Panel also contains mixed-case letters. However, lowercase letters in an ordinary token are folded to uppercase by the SQL Preprocessor. Lowercase letters in a delimiter token remain lowercase.
- You must enter a semicolon after each complete SQL statement.

After placing your SQL statement or statements in the numbered line area on the Source Panel, you can perform the following:

1. Press PF9 to execute the source member. CA Datacom Datadictionary responds with the Output Panel which shows the results of the SQL processing.
2. Press PF2 (END) to return to the SQLMAINT menu panel where you can select a function. If you have refreshed the panel by pressing ENTER or a PF key, your entries are saved in the Source Member.
3. Press the CLEAR key to return to the Interactive SQL Service Facility Panel where you can choose another option. If you have refreshed the panel by pressing ENTER or a PF key, your entries are saved in the Source Member.

**Step 6**

When you execute the source member, you receive a Output Panel. The following example shows the Output Panel displayed after a CREATE TABLE statement was executed.

```

=>
=>
=>
----- >>>
Interactive SQL Service Facility                               SQLMAINT

                                Output Panel
                                EDIT      Member= $DDSQL
                                Description: CREATE A SCHEMA
-----
===== T O P =====
000001 (OUTPUT CREATED FROM SOURCE MEMBER $DDSQL )
000002 CREATE TABLE DEPTTBL
000003         (DEPTNO CHAR(2) NOT NULL,
000004         DEPTNAME CHAR(24) NOT NULL);
000005 -----+-----+-----+-----+-----+-----+-----+
000006 STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
000007 -----+-----+-----+-----+-----+-----+
000008 SQL STATEMENTS ASSUMED TO BE BETWEEN COLUMNS 1 AND 72
000009 NUMBER OF INPUT RECORDS READ IS 0003
PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD   PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT     PF12=ALTERNATE

```

To exit from the Output Panel, you can perform one of the following.

- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press PF12 to display the Source Panel.
- Press the CLEAR key to return to the Interactive SQL Service Facility Panel.

## Using Commands

In the Interactive SQL Service Facility, you can make your selections through menu and prompter panels or by entering commands in the command lines of the panels.

**Note:** For complete information on using CA Datacom Datadictionary online, see the *CA Datacom Datadictionary User Guide* or *CA Datacom Datadictionary Online Reference Guide*.

The abbreviated command syntax for the menu options is displayed on the panels to help you learn the commands easily. The full and abbreviated command syntax for functions performed by the prompter panels are displayed on those panels.

You can skip the prompter panel by entering these commands. However, if the complete command is longer than the 76 spaces available in a command line, you must use the prompter panel. A command cannot be continued to another line.

You can display HELP panels for most commands by entering HELP, followed by the command. The HELP panel explains the functions and options for using that particular command.

To issue commands to CA Datacom Datadictionary, enter the command in the command region (region 0) at the top of the panel and then press Enter. If a command is only valid in a specific processing mode, you must select the appropriate mode before you enter the command. You can select a processing mode by entering the appropriate option on the Datadictionary Mode Select Panel or by entering the SET MODE command with the selected mode identifier.

Except where noted, you can enter more than one command on a panel as shown in the following examples. You can change the number of command lines displayed with the SET CMD LIN command. See the *CA Datacom Datadictionary Online Reference Guide* for details.

- You can type each command on a separate line. For example:

```
=> SET MODE SQL
=> EDIT SQL
=>
```

-----

- Or, you can type several commands on the same line and separate them with a semicolon, the default delimiter. You can change the delimiter with the SET CMD DLM command. See the *CA Datacom Datadictionary Online Reference Guide* for details. For example:

```
=> SET MODE SQL;EDIT SQL
=>
=>
```

When you enter only a portion of a command, CA Datacom Datadictionary presents the prompter panel or menu for that function with the information you have filled in on the panel.

Remember that on a prompter panel, you must do the following:

- Supply all required entries.
- Supply any desired optional entries.
- Press Enter or PF9 (APPLY) to continue processing.

You can use the following general commands in the Interactive SQL Service Facility. See the *CA Datacom Datadictionary Online Reference Guide* for explanations of the commands in the following list.

- BOTTOM
- COMBINE
- END
- HELP
- INPUT
- MENU
- OFF
- POSITION
- PROCESS
- SET
- SPLIT
- STATUS
- SRB
- SRF
- TIME

## Commands Specifically for Use in the Interactive SQL Service Facility

The following CA Datacom Datadictionary online commands are specifically for the Interactive SQL Service Facility. You can use CA Datacom Datadictionary online to perform administrative functions such as setting the session default AUTHID, deleting plans, and rebinding plans. These functions are described in Performing SQL Administrative Tasks.

**Note:** The following commands are valid for maintenance in the Interactive SQL Service Facility mode in CA Datacom Datadictionary only. They do not perform SQL functions.

### ALTERNATE Command

The ALTERNATE command allows you to switch the display to the Source Panel or the Output Panel when both source and output members are entered on a prompter panel. The ALTERNATE command is also available with the PF12 key.

### COPY SQL Command

You can duplicate a source member with the COPY SQL command. Output members cannot be copied. Use the following format for this command. The abbreviated command syntax is COP.

```
on  
COPY SQL /old-source-name[, [new-source-name][, person]]/  
off
```

**old-source-name**

*(Required)* Enter the name of the existing source member.

**Valid Entries:**

1 to 8 characters

**Default Value:**

(No default)



**new-source-name**

*(Optional)* Enter a valid source member name that is unique for the specified or current default person.

**Valid Entries:**

1 to 8 characters

**Default Value:**

If the person is not specified with this command or is the same as the person currently signed on to CA Datacom Datadictionary, there is no default and you must enter a new source name.

If the person is specified, and is not the person currently signed on to CA Datacom Datadictionary, this field defaults to the old source name.

**,person**

*(Optional)* Enter the name of the person who created the existing source member. If you leave this field blank, CA Datacom Datadictionary uses the name of the person currently signed on.

**Valid Entries:**

The first 18 characters of the 1- to 32-character occurrence name

**Default Value:**

PERSON occurrence name associated with the person currently signed on to CA Datacom Datadictionary

CA Datacom Datadictionary automatically copies the member indicated by the COPY command, and displays the SQL Member List Panel to display the source members that currently exist. A message also displays on the message line indicating the status of the copy function.

After copying the source member, you can perform one of the following:

- Enter another source member name to copy on the SQL Member List Panel.
- Enter another SQL command on the command line.
- Press PF2 (END) to return to the SQLMAINT Panel, where you can select another maintenance function.
- Press CLEAR to return to the Interactive SQL Service Facility Panel to perform another SQL function.

## DELETE SQL Command

You can use the DELETE SQL command to delete a source or output member.

**Note:** Use the SQL DROP statement to delete SQL-accessible tables, views, or synonyms. See [Deleting SQL Objects](#) (see page 431) and [DROP](#) (see page 725).

Use the following format for the DELETE SQL command. The abbreviated command syntax is DEL.

**on**

```
DELETE SQL /[source-member-name] [,output-member-name]/
```

**off**

**source-member-name**

*(Optional)* Enter the name of the source member that you are deleting.

**Valid Entries:**

1 to 8 characters

**Default Value:**

(No default)

**,output-member-name**

*(Optional)* Enter the name of the output member that you are deleting.

**Valid Entries:**

1 to 8 characters

**Default Value:**

(No default)

After you enter the DELETE SQL command, CA Datacom Datadictionary displays the SQL Member Selection Criteria Fill-in Panel with the source and/or output member name displayed. Press Enter on this panel to delete the member(s).

After successfully deleting the members, you can perform one of the following functions:

- Enter another source and/or output name to delete.
- Enter another SQL command on the command line.
- Press PF2 (END) to return to the SQLMAINT Panel.
- Press CLEAR to return to the Interactive SQL Service Facility Panel to perform another function.

## DISPLAY SQL Command

Use the DISPLAY SQL command to display a specific source or output member, or display a list of the members for the person currently signed on to CA Datacom Datadictionary. To display a specific source or output member, use the following format for this command. The abbreviated command syntax is DIS.

**on**

DISPLAY SQL / [source-member-name] [, output-member-name] [, person-occurrence-name] /  
**off**

### **source-member-name**

*(Optional)* Enter the name of the member that contains the SQL statements you want to display.

#### **Valid Entries:**

1 to 8 characters

#### **Default Value:**

(No default)

### **,output-member-name**

*(Optional)* Enter the name of the member that contains the results of executing a source member.

#### **Valid Entries:**

1 to 8 characters

#### **Default Value:**

(No default)

### **,person-occurrence-name**

*(Optional)* Enter the name of the person who created the source member.

#### **Valid Entries:**

1- to 32-character PERSON occurrence-name

#### **Default Value:**

PERSON occurrence-name associated with the person currently signed on to CA Datacom Datadictionary

To display a list of all source and output members for a specific PERSON occurrence, use the following command. The abbreviated command syntax is DIS.

**on**  
DISPLAY SQL LIST [person-occurrence-name]  
**off**

**person-occurrence-name**

*(Optional)* Enter the name of the person who created the source members you want to display.

**Valid Entries:**

1- to 32-character PERSON occurrence name

**Default Value:**

PERSON occurrence name associated with the person currently signed on to CA Datacom Datadictionary

After entering the DISPLAY SQL LIST command, CA Datacom Datadictionary displays the SQL Member List Panel. If the display is longer than the screen, you can use the scroll commands or PF keys to move backward or forward through the display.

You can enter the following SQL margin commands on the SQL Member List Panel: COP or CPY, DEL, DIS, and EDT. See [Using Margin Commands](#) (see page 376).

## EDIT SQL Command

Use the EDIT SQL command to create or modify a source member. Enter the following command on the command line on a panel. The abbreviated command syntax is EDT.

**on**  
EDIT SQL /[source-member-name][,output-member-name][,description]/  
**off**

**source-member-name**

*(Optional)* Enter the name of the member containing the SQL statements you want to edit.

**Valid Entries:**

1 to 8 characters

**Default Value:**

(No default)

**,output-member-name**

*(Optional)* Enter the name of the member containing the results of executing a source member.

**Valid Entries:**

1 to 8 characters

**Default Value:**

(No default)

**,description**

*(Optional)* You can enter the 32-character description of the source member.

**Valid Entries:**

1 to 32 characters

**Default Value:**

(No default)

After you enter the EDIT command, CA Datacom Datadictionary displays the Source Panel where you can perform one of the following functions:

- Edit the SQL statement(s) or enter additional SQL statements. Place a semicolon after each complete statement.
- Press PF9 (EXECUTE) to execute the source member. CA Datacom Datadictionary responds with the Output Panel which shows the results of SQL processing.
- Press PF2 (END) to obtain an SQLMAINT Panel where you can select the EDIT function and then specify another source member.
- Press CLEAR to return to the Interactive SQL Service Facility Panel to choose another function.

## EXECUTE Command

The EXECUTE command submits the source member for execution. It is also available with the PF9 key.

## REBIND Command

The REBIND command is valid in this mode only. It accesses the SQLADMIN panel where you enter the information necessary to rebind plans. See [Rebinding a Plan](#) (see page 457) for details.

## SCROLL Command

Use the SCROLL command to move forward and backward through the lines of a panel that you cannot display all at one time on a single screen. Use the following format for this command. The abbreviated command syntax is SCR. If your completed command exceeds the 76 spaces available in one command line, you must use the prompter panel to issue the command.

```
ON
SCROLL [option]
OFF
```

If you do not add a keyword with the SCROLL command, CA Datacom Datadictionary scrolls the display forward the set number of lines in a region with the last line of the previous display at the top of the new display. You can also enter this command with the PF8 key.

You can add the following optional keywords to the SCROLL command:

Option	Action
+	Scroll forward the set number of lines in a region. (You can also use the PF8 key, the SCROLL command or the SRF command for this function.)
-	Scroll backward the set number of lines in a region. (You can also use the PF7 key or the SRB command for this function.)
+nnn	Scroll forward nnn number of lines in a region.
-nnn	Scroll backward nnnn number of lines in a region.
B BACKWARD	Scroll backward the set number of lines in a region. (You can also use the PF7 key, the SCROLL command, or the SRB command for this function.)
BOT BOTTOM	Scroll to bottom of the list. (You can also use the BOTTOM command for this function.)
F FORWARD	Scroll forward the set number of lines in a region. (You can also use the PF8 key, the SCROLL command, the SCROLL + command, or the SRF command for this function.)
LEF LEFT	Scroll left one full screen of an output member display. (You can also use the PF10 key.) The information in the output member can be 228 columns wide (including data, column headings, and spaces between columns).
LEF nn LEFT nn	Scroll left a specified number of columns on an output member display, where nn is the number of columns. You can also optionally enter COL after nn.

Option	Action
RIG RIGHT	Scroll right one full screen of an output member display. (You can also use the PF11 key.) The output member can be up to 228 columns (including data, column headings, and spaces between columns).
RIG nn RIGHT nn	Scroll right a specified number of columns on an output member display, where nn is the number of columns. You can also optionally enter COL after nn.
TOP	Scroll to the top of the list. (You can also use the TOP command for this function.)

## Using Line Commands

You can use line commands to scroll through, insert, copy, move, and delete lines in repeating groups on Source Panels. These commands are not related to the online work queue commands which are called margin commands. See [Using Margin Commands](#) (see page 376) for a list of these commands.

The numerical factor described with the following commands can be either left or right of the command depending on the site option selected. For example, to insert 3 lines, you enter either I3 or I3I depending on the format established at your site in the Datadictionary System Resource Table (SRT). See the *CA Datacom/DB Database and System Administration Guide* for information.

Line Command	Action
*	Scroll the display until this line is at the top of the panel.
*+nnn	Scroll the display until the line that is nnn lines after this line is at the top of the panel.
*-nnn	Scroll the display until the line that is nnn lines before this line is at the top of the panel.
A	Designate the location after which lines are to be copied or moved.
B	Designate the location before which lines are to be copied or moved.
C	Copy a line. Use A or B to designate location where the line is to be copied.
CC	Copy a block of lines. CC must be entered on the first line and the last line of the block. Use A or B to designate location where block is to be copied.
D	Delete a line.

<b>Line Command</b>	<b>Action</b>
DB	Delete all lines from this line through the bottom line.
dd	Delete a block of lines. DD must be entered on the first line and the last line of the block.
DT	Delete all lines from this line through the top line.
I	Insert a line after the line where the command is entered.
IB	Insert a blank line before the first line.
In	Insert n number of blank lines after the line where the command is entered.
M	Move a line. Use A or B to designate where the line is to be moved.
MM	Move a block of lines. MM must be entered on the first line and the last line of the block. Use A or B to designate where the block is to be moved.
R	Repeat the line.
Rn	Repeat the line n times after itself.
RR	Repeat a block of lines. RR must be entered on the first line and the last line of the block. The lines are repeated after the last line of the designated block.

## Using Margin Commands

CA Datacom Datadictionary online provides an online work queue that allows you to store specific functions for processing in sequence. You activate this function by placing margin commands in the line numbers of display panels. The online work queue is session dependent and is deleted when you exit CA Datacom Datadictionary online.

After entering margin commands, you can either:

- Press Enter or any scrolling PF key to refresh the screen and enter additional margin commands as desired.
- Press PF4 or enter the PROCESS command to bring the first activity from the online work queue.



You can enter the following margin commands in the line numbers on the SQL Member List Panel:

Margin Commands	Restriction and Action
COP or CPY	Enter for a source member only. Displays the SQL Member Selection Criteria Fill-in Panel for COPY with the source member name and person who created the source member filled in. See <a href="#">Copying Source Members</a> (see page 393).
DEL	Enter for a source or output member that was created by the person currently signed on to CA Datacom Datadictionary. Displays the SQL Member Selection Criteria Fill-in Panel for DELETE with the member name and description filled in. See <a href="#">Deleting Source and Output Members</a> (see page 395).
DIS	Enter for a source or output member. Displays the SQL Member Selection Criteria Fill-in Panel for DISPLAY with the member name and person name filled in. You can then enter the source or output member name, whichever was not selected by the margin command. See <a href="#">Displaying Source and Output Members</a> (see page 388).
EDT	Enter for a source member that was created by the person currently signed on to CA Datacom Datadictionary. Displays the SQL Member Selection Criteria Fill-in Panel for EDIT with the source member name and description filled in. See <a href="#">Editing and Executing Source Members</a> (see page 380).

## Using PF Keys

The PF and PA keys are assigned the following functions in the Interactive SQL Service Facility. Keys not mentioned are ignored by CA Datacom Datadictionary.

Key	Equivalent Command and/or Action
PF1	HELP - displays current HELP panel.
PF2	END - displays the previous level panel.
PF3	SPLIT - displays two sessions simultaneously, if your terminal displays more than 30 lines. Use the COMBINE command to end the session added with PF3 (SPLIT).
PF4	PROCESS - retrieves and executes the next occurrence and function from the online work queue. The abbreviated form is PRO. See the <i>CA Datacom Datadictionary Online Reference Guide</i> .

<b>Key</b>	<b>Equivalent Command and/or Action</b>
PF5	TOP - scrolls the display on variable line panel to place the first line at the top of the repeating group section.
PF6	BOTTOM - scrolls the display on a variable line panel to place the last line at the bottom of the repeating group section.
PF7	BACKWARD - scrolls backward the default number of lines in the repeating group section of a variable line panel.
PF8	FORWARD - scrolls forward the default number of lines in the repeating group section of a variable panel.
PF9	EXECUTE - submits for processing the SQL statements entered on the Source Panel and activates the next logical panel. The abbreviated form is EXE.
PF10	LEFT - scrolls left the default number of columns when the displayed member has more than 72 columns. The output member can be up to 228 columns wide (including data, column headings, and spaces between columns).
PF11	RIGHT - scrolls right the default number of columns when the displayed member has more than 72 columns. The output member can be up to 228 columns wide (including data, column headings, and spaces between columns).
PF12	ALTERNATE - alternately displays the Source Panel and the Output Panel when both a source member name and an output member name were entered on a prompter panel.
PA1	Refreshes the screen.
PA2	Displays PF/PA key assignments.
CLEAR	Displays the Interactive SQL Service Facility menu.
ENTER	Performs the following functions: <ul style="list-style-type: none"><li>■ On delete panels: it deletes the selected source or output member.</li><li>■ On display panels: it stores the margin and line commands and refreshes the screen. This allows you to specify additional margin commands.</li><li>■ On variable line maintenance panels: it executes any line commands you type in the line numbers and refreshes the screen.</li><li>■ On prompter panels and menu panels: it displays the next panel when there is a series of panels.</li></ul>

## Maintaining Source and Output Members

Use the SQLMAINT option of the Interactive SQL Service Facility to maintain the CA Datacom Datadictionary source and output members. This section explains how to edit, execute, display, delete, and copy the source members in CA Datacom Datadictionary. It also explains how you display and delete output members using this option.

The source and output members are saved by CA Datacom Datadictionary in a Virtual Library System (VLS) member (the VLS is a library access method used to store panels, message members, control blocks, and user programs). The VLS is one of the CA IPC. The name of the VLS member is specified in the System Resource Table parameter DDOLSQL= (in the DDSYSTBL macro).

**Note:** For more information about DDOLSQL=, see the *CA Datacom/DB Database and System Administration Guide*. For more information on the Virtual Library System and the VLS Utility (VLSUTIL) that enables you to modify and maintain the VLS files at your site, see the *CA IPC Implementation Guide*.

When you select option 1 (SQLMAINT) on the Interactive SQL Service Facility panel, CA Datacom Datadictionary displays the following panel:

```

=>
=>
=>

-----
Interactive SQL Service Facility                                SQLMAINT
Enter desired option number ==> __ (There are 05 options on the menu)

 1. EDIT                Edit/Execute SQL members
 2. DISPLAY             Display source/output members
 3. DELETE              Delete source/output members
 4. COPY                Copy source members
 5. END                 End SQLMAINT processing

```

Enter the number for the option you want to use and press Enter. CA Datacom Datadictionary displays the SQL Member Selection Criteria Fill-in Panel. The panel displays the option at the left of the panel and the fields needed for processing.

## Editing and Executing Source Members

Use option 1 (EDIT) on the SQLMAINT Panel to create, modify, or execute a member containing SQL statements. CA Datacom Datadictionary displays a SQL Member Selection Criteria Fill-in Panel as in the following example: The panel displays the EDIT option you requested on the left side of the panel, and provides fields for you to enter the names of the source and output members and a description.

You can also use the EDT SQL command in the command area in the Interactive SQL Service Facility. For details about entering this command, see [EDIT SQL Command](#) (see page 372).

```

=>
=>
=>

-----
Interactive SQL Service Facility                                SQLMAINT
                    SQL Member Selection Criteria Fill-in      S01F

EDIT   SQL / _____ , _____ , _____ /
EDT    (source) (output) (description)
        (name ) (name )

NOTE: If no source/output member is entered the default is $DDSQL.

PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD   PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT     PF12=ALTERNATE
  
```

Complete your specifications on the SQL Member Selection Criteria Fill-in Panel according to the following descriptions:

**source name**

*(Optional)* Enter the name of the member that contains the SQL statements. Use alphanumeric characters in the name. It cannot contain special characters or embedded blanks.

When the member name you enter already exists for the current user, the next panel displays the existing source member for modification. If the member does not exist, a new member is created. If you leave this field blank, the default name is used.

**Valid Entries:**

1 to 8 characters

**Default Value:**

\$DDSQL

**output name**

*(Optional)* Enter the name of the member that contains the results of executing the source member. Use alphanumeric characters in the name. It cannot contain special characters or embedded blanks.

For identification convenience, use the same name for both the source and output members. If the output member name you enter already exists, CA Datacom Datadictionary replaces the previous output member upon execution. If you leave this field blank, the default name is used.

**Valid Entries:**

1 to 8 characters

**Default Value:**

\$DDSQL

**description**

*(Optional)* You can enter up to 32 characters and embedded blanks to describe your source members. The same description is displayed with the output member.

**Valid Entries:**

1 to 32 characters

**Default Value:**

(No default)

**Source Panel**

After you complete the SQL Member Selection Criteria Fill-in Panel, and press Enter, CA Datacom Datadictionary displays the Source Panel.

```

=>
=>
=>
-----
Interactive SQL Service Facility                               SQLMAINT
                                     Source Panel                               S01S
EDIT   Member: $DDSQL   Output Line Limit: 01000
      Person: JONES
      Current Authid: JONES
      Description: CREATE TABLE DEPTTBL
-----
===== T O P =====
000001
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD  PF8=FORWARD
PF9=EXECUTE   PF10=LEFT     PF11=RIGHT    PF12=ALTERNATE
  
```

Information you provided on the SQL Member Selection Criteria Fill-in Panel displays in the top portion of this panel. In this example, a CREATE TABLE SQL statement is being executed. See the following explanation of the fields that appear on this panel.

**Member:**

The source member name you entered on the SQL Member Selection Criteria Fill-in Panel or the default member name \$DDSQL.

**Output Line Limit:**

*(Optional)* The maximum number of lines you receive on the output panel. Even though a portion of the output member information is displayed, the entire statement is executed.

If you want the output to display more than 1000 lines, we recommend that you enter the SQL statements in an application program, or see the person responsible for administering CA Datacom/DB at your site.

**Valid Entries:**

Up to 1000

**Default Value:**

1000

**Person:**

The PERSON occurrence associated with the user ID entered on the CA Datacom Datadictionary Signon Panel.

**Current Authid:**

One of the following is displayed:

- The AUTHID related to the current user
- The AUTHID established for the session with the SET AUTHID option.

**Description:**

The description you entered on the SQL Member Selection Criteria Fill-in Panel.

**numbered line(s)**

If this is a new source member, you receive one numbered blank line where you enter your SQL statement(s). See the instructions in [Submitting SQL Statements](#) (see page 357) to enter the SQL statements you can execute in the Interactive SQL Service Facility. End each statement with a semicolon.

Use line commands to insert the number of text lines needed to enter the statement(s). See [Using Line Commands](#) (see page 375).

After completing the statement, press Enter. CA Datacom Datadictionary displays the SQL statement as it was entered.

```

=>
=>
=>

-----
Interactive SQL Service Facility                                SQLMAINT
                                     Source Panel                S01S
EDIT   Member: $DDSQL   Output Line Limit: 01000
        Person: JONES
        Current Authid: JONES
        Description: CREATE TABLE DEPTTBL
-----
===== T O P =====
000001 CREATE TABLE DEPTTBL
000002     (DEPTNO CHAR(2) NOT NULL,
000003     DEPTNAME CHAR(24) NOT NULL);
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD  PF8=FORWARD
PF9=EXECUTE   PF10=LEFT     PF11=RIGHT    PF12=ALTERNATE
  
```

At this point in processing, you can perform one of the following functions:

- Enter additional SQL statements. Place a semicolon after each complete statement.
- Press PF9 (EXECUTE) to execute the source member. CA Datacom Datadictionary responds with the Output Panel which shows the results of SQL processing.
- Press PF2 (END) to obtain an SQLMAINT Panel where you can select the EDIT function and then specify another source member.
- Press CLEAR to return to the Interactive SQL Service Facility Panel to choose another function.

### Output Panel

When you execute the statement, you receive the Output Panel. The following example is output for the CREATE TABLE statement for the DEPTTBL table.

```
=>
=>
=>

----- >>>
Interactive SQL Service Facility                               SQLMAINT

                        Output Panel                          S010
                        EDIT      Member= $DDSQL
                        Description: CREATE A SCHEMA
-----

===== T O P =====
000001 (OUTPUT CREATED FROM SOURCE MEMBER $DDSQL )
000002 CREATE TABLE DEPTTBL
000003      (DEPTNO CHAR(2) NOT NULL,
000004      DEPTNAME CHAR(24) NOT NULL);
000005 -----+-----+-----+-----+-----+-----+
000006 STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
000007 -----+-----+-----+-----+-----+
000008 SQL STATEMENTS ASSUMED TO BE BETWEEN COLUMNS 1 AND 72
000009 NUMBER OF INPUT RECORDS READ IS 0003
PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD  PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT    PF12=ALTERNATE
```



When the display is longer than the screen, you can use the scroll commands or PF keys to display the additional information. Another example panel showing an error message and return code follows the description of the fields on this panel.

**Note:** Certain SQL statements, such as SELECT, return information that extends beyond the right column of a display. Use the scroll commands or PF keys to display this information. However, the returned data is truncated when the information on a line exceeds 228 bytes. The source and output members are stored in the Virtual Library System (VLS) and the VLS has a record limit of 240 bytes, of which 12 bytes are used internally.

The top portion of the panel displays the member name and description you entered on the Source Panel. For this example, the text portion of the panel displays the contents of the output member the following segments.

Line numbers referenced are *only* valid as illustration for this example, the output varies with *each* SQL statement you execute.

**Line 1**

The source member that created the output is displayed.

**Lines 2-4**

The SQL statement(s) as entered and the result(s) of the executing statement(s) are displayed.

**Line 6**

A message stating the success or failure of the executed statement is displayed. An SQLCODE of 0 indicates the statement execution was successful. When you execute a SELECT statement, an SQLCODE of 100 also indicates successful execution. SQL return codes are listed in the *CA Datacom/DB Message Reference Guide*.

**Remaining Lines**

Additional support information is included: the number of columns CA Datacom Datadictionary uses to process the SQL statements, the number of input records read, the number of SQL statements processed.

If the returned data exceeds the 228-byte per line limit, the support information includes: RECORD LENGTH EXCEEDS MAXIMUM, DATA MAY BE TRUNCATED.

### Error Messages Displayed

When an error in processing is encountered, more information is displayed containing the appropriate return codes, and whether the error was encountered in processing CA Datacom Datadictionary or CA Datacom/DB.

```
⇒
⇒
⇒

----- >>>
Interactive SQL Service Facility                               SQLMAINT

                               Output Panel                     S010
EDIT      Member= CSCHEMA
          Description: CREATE A SCHEMA
-----
===== T O P =====
000001 (OUTPUT CREATED FROM SOURCE MEMBER CSCHEMA )
000002 CREATE SCHEMA AUTHORIZATION JONES;
000003 -----+-----+-----+-----+-----+-----+
000004 ERROR OCCURRED EXECUTING A PLAN  -0118(MAAE)           (QEXEI)
000005 -----+-----+-----+-----+-----+-----+
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT   PF12=ALTERNATE
```

In this example, the -0118 SQL return code indicates that an error was detected by the CA Datacom Datadictionary Service Facility (DSF) while attempting to process the statement. The MAAE return code indicates with M that the return code is from a module of CA Datacom Datadictionary and with AAE (a DSF return code) that the AUTHORIZATION occurrence already exists in CA Datacom Datadictionary.

Error messages appear in the following general format:

**ERROR OCCURRED cccccccccccccccccc - nnnn(xyxy) (bbbbbb)**

Indicates an error was encountered during execution of the statement.

where:

**cccccccccccccccccccc**

Indicates where the error occurred. In the previous example, an error occurred executing the plan.

**nnnnn(xyyy)**

The nnnnn string indicates that an error occurred during CA Datacom/DB or CA Datacom Datadictionary execution. The -0118 in the previous example, indicates an error occurred in executing CA Datacom Datadictionary. A return code of -0117 indicates an error occurred in executing CA Datacom/DB.

The (xyyy) represents the return codes. The x indicates the type of CA Datacom Datadictionary return code and the yyy represents the CA Datacom Datadictionary Service Facility (DSF) return code.

See [SQL Error Handling](#) (see page 289) for information.

**(bbbbbb)**

These characters represent an internal SQL code.

Return codes for SQL and other related CA Datacom products are listed in the *CA Datacom/DB Message Reference Guide*.

The output member can be up to 228 columns wide. To view the columns beyond the displayable screen area, use the SCROLL commands or PF11 to scroll right and PF10 to scroll left. See [SCROLL Command](#) (see page 374) for more instructions.

To exit from the Output Panel, you can do one of the following:

- Press PF2 (END) to obtain an SQLMAINT Panel where you can select the EDIT function and then specify another source member.
- Press PF12 (ALTERNATE) to display the Source Panel.
- Press CLEAR to return to the Interactive SQL Service Facility Panel.

You can execute the statements in the source member again, or use the member to create other source members. Output members can be displayed at a later time or deleted. See the following sections for displaying, copying, and deleting source members, and for displaying and deleting output members.

## Displaying Source and Output Members

You can display the contents of a specific source or output member or a list of all members created by a person. The person can be a named person or the person currently signed on to CA Datacom Datadictionary.

### Display Specific Member

Use one of the following to display a specific member.

- Select option 2 (DISPLAY) on the SQLMAINT Panel and complete the entries on the SQL Member Selection Criteria Fill-in Panel.
- Enter the full DISPLAY SQL command. See [DISPLAY SQL Command](#) (see page 371) for information on entering this command.
- Leave the panel blank and press Enter to receive the list of members. You can then enter the DIS margin command next to the member you want to display.
- Enter only DISPLAY SQL to receive the panel and press Enter to receive the list of members. You can then enter the DIS margin command next to the member you want to display.

### Display Member List

Use one of the following to display a specific member.

- Leave the SQL Member Selection Criteria Fill-in Panel blank and press Enter to display a list of all source and output members.
- Specify only the person on the SQL Member Selection Criteria Fill-in Panel to display a list of all source and output members for that person.
- Enter the DIS SQL LIST command in the command line of a panel to obtain the list for the person currently signed on or include a valid PERSON occurrence name in the command to obtain the list for that person. See [DISPLAY SQL Command](#) (see page 371) for information on entering this command.

If you select option 2 (DISPLAY) on the SQLMAINT Panel, CA Datacom Datadictionary displays a SQL Member Selection Criteria Fill-in Panel. The panel displays the function you are requesting on the left side of the panel.

```

=>
=>
=>
-----
Interactive SQL Service Facility                                SQLMAINT
                    SQL Member Selection Criteria Fill-in      S02F

DISPLAY SQL / _____ , _____ , _____ /
DIS       (source) (output) (person)
          (name )  (name )

NOTE: Press Enter to obtain a list of source/output members.

PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD   PF8=FORWARD
PF9=EXECUTE   PF10=LEFT     PF11=RIGHT     PF12=ALTERNATE
  
```

Complete your specifications on the SQL Member Selection Criteria Fill-in Panel according to the following descriptions if you are displaying a specific source and/or output member.

**source name**

*(Optional)* Enter the name of the member that contains the SQL statements you want to display. Leave this field blank if you want to display the output member only. Leave both fields blank to obtain the SQL Member List Panel.

**Valid Entries:**

1 to 8 characters

**Default Value:**

(No default)

**output name**

*(Optional)* Enter the name of the member that contains results of executing a source member. Leave this field blank if you want to display the source member only. Leave both fields blank to obtain the SQL Member List Panel.

**Valid Entries:**

1 to 8 characters

**Default Value:**

(No default)

**person**

*(Optional)* Enter the name of the person who created the source and output members. Leave this field blank to obtain the SQL Member List for the person currently signed on to CA Datacom Datadictionary.

**Valid Entries:**

The first 18 characters of the 1- to 32-character PERSON occurrence name

**Default Value:**

PERSON occurrence name associated with the person currently signed on to CA Datacom Datadictionary

After completing the SQL Member Selection Criteria Fill-in Panel, press Enter to display the Source or Output Panel. If you have left the source and output member names blank, a list of all current source and output members for the specified person displays, as in the following example:

```

=>
=>
=>
-----
Interactive SQL Service Facility                                SQLMAINT
                                SQL Member List                S01N
                                Person: JONES

      Name      Type Description                                Updated  Lines
=====
000001 $DDSQL  0  CREATE MY SCHEMA                                112099  00004
000002 DEPTTBL  0  DEPARTMENT ORGANIZATION TABLE                112099  00011
000003 $DDSQL  S  CREATE MY SCHEMA                                112099  00001
dis004 DEPTTBL  S  DEPARTMENT ORGANIZATION TABLE                112099  00006
=====
                                B O T T O M
-----

PF1=HELP      PF2=END      PF3=SPLIT    PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD PF8=FORWARD
PF9=EXECUTE   PF10=LEFT   PF11=RIGHT   PF12=ALTERNATE
    
```

When the display is longer than the screen, you can use the scroll commands or PF keys to move backwards and forwards through the display. Another panel to display the additional fields follows the description of the fields on this panel.

The following information is displayed for each member.

**Name**

The name of the source or output member.

**Type**

Identifies whether the member is a source (S) or output (O) member.

**Description**

The description entered for the source member when the member was created.

**Updated**

The date the member was updated in month, day, and year format (mmddyy).

**Lines**

The number of lines in the member.

The members are listed in alphabetical order by type with the output members first.

**Example**

The previous SQL Member List Panel shows an example of entering the DIS margin command on line 004. See [Using Margin Commands](#) (see page 376) for more information on these commands. After entering margin commands, you can perform either of the following:

- Press Enter or any scrolling PF key to refresh the screen and enter additional margin commands as desired.
- Press PF4 or enter the PROCESS command to bring the first activity from the online work queue. In this example, CA Datacom Datadictionary displays the SQL Member Selection Criteria Fill-in Panel for DISPLAY where you can enter the associated source or output member name, or change the displayed names.

After you have completed any entries on the SQL Member Selection Criteria Fill-in Panel, press Enter to display the source and/or output members. CA Datacom Datadictionary always displays the source members first, if you have entered a source member name on the prompter panel.

The following panel illustrates the results of using the DIS margin command to display the source member for DEPTTBL.

```
=>
=>
=>
-----
Interactive SQL Service Facility                                SQLMAINT
                                Source Panel                    S01S
EDIT      Member: DEPTTBL   Output Line Limit: 01000
          Person: JONES
          Current Authid: JONES
          Description: CREATE TABLE DEPTTBL
-----
===== T O P =====
000001 CREATE TABLE DEPTTBL
000002      (DEPTNO CHAR(2) NOT NULL,
000003      DEPTNAME CHAR(24) NOT NULL);
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD  PF8=FORWARD
PF9=EXECUTE   PF10=LEFT   PF11=RIGHT    PF12=ALTERNATE
```

To exit from either the Source or Output Panels, you can do one of the following:

- Press PF12 (ALTERNATE) to obtain the associated source or output member you named on the prompter panel.
- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press CLEAR to return to the Interactive SQL Service Facility Panel to select another function.



## Copying Source Members

You can use the copy function of the Interactive SQL Service Facility to duplicate a source member. To copy a source member, either:

- Select option 4 (COPY) on the SQLMAINT Panel.
- Enter the CPY or COP margin command on the SQL Member List Panel. See [Displaying Source and Output Members](#) (see page 388) for details.
- Enter the COP SQL command on the command line in Interactive SQL Service Facility. See [COPY SQL Command](#) (see page 368) for details.

If you select option 4 on the SQLMAINT Panel, CA Datacom Datadictionary displays the SQL Member Selection Criteria Fill-in Panel with the option you are requesting on the left of the panel. Use this panel to identify the source member being copied.

```

=>
=>
=>
-----
Interactive SQL Service Facility                                SQLMAINT
                                SQL Member Selection Criteria Fill-in          S02F

COPY   SQL / _____ , _____ , _____ /
CPY    (old  ) , (new  ) , (person - owner of old source member)
        (source) (source)
        (name ) (name )

PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD   PF8=FORWARD
PF9=EXECUTE   PF10=LEFT     PF11=RIGHT     PF12=ALTERNATE
  
```

Complete your specifications on the SQL Member Selection Criteria Fill-in Panel according to the following descriptions.

**old source name**

*(Required)* Enter the name of the existing member. If you receive this panel after entering a margin command on the SQL Member List Panel, the existing source member name is filled in.

**Valid Entries:**

1 to 8 characters

**Default Value:**

(No default)

**new source name**

*(Optional)* Enter a valid source member name that is unique for the specified or current default person.

If the person is not specified with this command or is the same as the person currently signed on to CA Datacom Datadictionary, you must enter a new source name.

If the person is specified, and is not the person currently signed on to CA Datacom Datadictionary, this field defaults to the old source-name.

**Valid Entries:**

1 to 8 characters

**Default Value:**

(No default)  
old source name

**person - owner of old source member**

*(Optional)* Enter the name of the person who created the existing source member. If you leave this field blank, CA Datacom Datadictionary uses the name of the person currently signed on.

**Valid Entries:**

The first 18 characters of the 1- to 32-character PERSON occurrence name

**Default Value:**

PERSON occurrence name associated with the person currently signed on to CA Datacom Datadictionary

Review your entry, and press Enter. CA Datacom Datadictionary automatically copies the member indicated and displays the SQL Member List Panel to display the members that currently exist. A message also displays on the message line indicating the status of the copy function.

```

=>
=>
=>
1-DD0L0000067I - SIFP - SUCCESSFUL SQL MEMBER COPY
-----
Interactive SQL Service Facility                                SQLMAINT
                                SQL Member List                S01N
                                Person= JONES

      Name      Type Description                                Updated  Lines
=====
000001 $DDSQL   0 CREATE MY SCHEMA                                112099  00004
000002 DEPTTBL  0 DEPARTMENT ORGANIZATION TABLE                112099  00010
000003 $DDSQL   S CREATE MY SCHEMA                                112099  00001
000004 DEPTTBL  S DEPARTMENT ORGANIZATION TABLE                112099  00003
000005 DEPTTBL2 S DEPARTMENT ORGANIZATION TABLE                112099  00003
=====
                                = B O T T O M =

PF1=HELP      PF2=END      PF3=SPLIT    PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT   PF12=ALTERNATE
  
```

After copying the source member, you can perform one of the following:

- Enter another source member name to copy.
- Press PF2 (END) to return to the SQLMAINT Panel to select another option.
- Press CLEAR to return to the Interactive SQL Service Facility Panel to perform another function.

## Deleting Source and Output Members

You can use the delete function of the Interactive SQL Service Facility to remove a source or output member. Only the source or output member is deleted with the delete function. The objects created by the statements in a member remain intact.

To delete a member, either:

- Select option 3 (DELETE) on the SQLMAINT Panel.
- Enter the DEL margin command on the SQL Member List Panel. See [Displaying Source and Output Members](#) (see page 388) for details.
- Enter the DEL SQL command in the command line in Interactive SQL Service Facility. See [DELETE SQL Command](#) (see page 370) for details.

If you select option 3 on the SQLMAINT Panel, CA Datacom Datadictionary displays the SQL Member Selection Criteria Fill-in Panel. Use this panel to identify the source member and/or the output member being deleted. The panel displays the option you are requesting on the left of the panel.

```
=>
=>
=>
-----
Interactive SQL Service Facility                                SQLMAINT
                                SQL Member Selection Criteria Fill-in      S01F

DELETE  SQL / _____ , _____ , _____ /
DEL      (source) (output) (description)
          (name ) (name )

NOTE: Press Enter to obtain a list of source/output members.

PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD   PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT     PF12=ALTERNATE
```

Complete your specifications on the SQL Member Selection Criteria Fill-in Panel according to the following descriptions if you are deleting a specific source and/or output member.

**source name**

*(Optional)* Enter the name of the member that you are deleting. If you obtain this panel after entering the DEL margin command on the SQL Member List Panel, the source member name and person name are displayed.

Leave this field blank if you are requesting a display of all members or if you only intend to delete an output member.

**Valid Entries:**

1 to 8 characters

**Default Value:**

(No default)

**output name**

*(Optional)* Enter the name of the member that you are deleting. If you obtain this panel after entering the DEL margin command on the SQL Member List Panel, the output member name and person name are displayed.

Leave this field blank if you are requesting a display of all members or if you only intend to delete a source member.

**Valid Entries:**

1 to 8 characters

**Default Value:**

(No default)

**description**

The description you entered on the SQL Member Selection Criteria Fill-in Panel displays.

Review your entries, and press Enter. CA Datacom Datadictionary automatically deletes the member(s) indicated and displays a message indicating the status of the deletion.

```

=>
=>
=>
1-DDOL000002I - S1FP - SUCCESSFUL DELETE OF MEMBER(S)
-----
Interactive SQL Service Facility                                SQLMAINT
                                SQL Member Selection Criteria Fill-in          S01F

DELETE  SQL / DEPTTBL , _____ , DEPARTMENT ORGANIZATIONAL TABLE /
DEL      (source) (output) (description)
          (name ) (name )

NOTE: Press Enter to obtain a list of source/output members.

PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD  PF8=FORWARD
PF9=EXECUTE   PF10=LEFT   PF11=RIGHT    PF12=ALTERNATE
  
```

If you do not know the source or output member name, press ENTER on the SQL Member Selection Criteria Fill-in Panel to display a listing of source and output members. Enter the DEL margin command in the line of the source and/or output members you want to delete.

Press Enter on the SQL Member Selection Criteria Fill-in Panel to delete the source and/or output members. If you have entered the DEL margin command, you can press PF4 (PROCESS) to delete the first member selected and proceed to the next selection or the SQL Member Selection Criteria Fill-in Panel.

After deleting the source and/or output members, you can perform one of the following:

- Enter another source and/or output name to delete.
- Press PF2 (END) to return to the SQLMAINT Panel.
- Press CLEAR to return to the Interactive SQL Service Facility Panel to perform another function.

# Chapter 17: Creating SQL Objects

---

This chapter contains the following topics:

In Interactive SQL Service Facility, you can use the SQL Data Definition Language to define the components of a relational database, alter a table definition and add or replace comments.

## **CREATE SCHEMA**

Defines the authorization ID (which is the name of the schema) and can also include table, view and privilege definitions.

## **CREATE TABLE**

Defines a base table and its columns. You must name the table and each column, and also specify the data type and length of each column. You can optionally specify if one or more columns are to have a unique value for each row in the table or other constraints, and the area where the table's data is stored.

## **ALTER TABLE**

Changes the definition of a base table, its columns and constraints.

## **CREATE INDEX**

Defines an index on one or more columns of a base table to improve performance of queries which reference that table.

## **CREATE VIEW**

Defines a view, or derived table, which can be based on one or more base tables, or even on other views. You can also specify search conditions which limit the rows appearing in the view.

## **CREATE SYNONYM**

Defines an alternative name for a table or a view. Synonyms are especially useful when referencing a table or view owned by another authorization ID since the definition includes the qualified name of the object.

## **COMMENT ON**

Adds or replaces text to the TABLE, COLUMN or VIEW occurrence in CA Datacom Datadictionary.

For information about the CA Datacom/DB implementation of support for CREATE PROCEDURE and CREATE TRIGGER/RULE statements, see [Procedures and Triggers](#) (see page 70) and [Datadictionary Support for Triggers and Procedures](#) (see page 86).

## Creating a Schema

A schema defines a user's SQL environment. You establish the schema and identify the name for the schema with the CREATE SCHEMA statement. Enter the CREATE SCHEMA statement in the numbered area of an EDIT Source Member Panel. See [How to Submit SQL Statements](#) (see page 358) for the steps to obtain the panel and an explanation of the fields on the panel. The schema consists of:

- An authorization ID, which is the name of the schema
- The SQL objects (tables, views, indexes, and synonyms) created by an identified user who uses or specifies the authorization ID when creating the objects
- The privileges the user gives to other users

If the CA Datacom/DB Security Facility is installed, you must be a global database owner to execute the CREATE SCHEMA statement.

**Note:** For information about global database owners, see the *CA Datacom Security Reference Guide*.

When you create a schema, CA Datacom Datadictionary defines the authorization ID as an AUTHORIZATION entity-occurrence. When the AUTHORIZATION occurrence is defined, the AUTH-USAGE attribute is assigned a value of S. This indicates that the occurrence is an SQL authorization ID. The AUTH-USAGE attribute is updateable by CA Datacom Datadictionary only, not by a user.

Only AUTHORIZATION occurrences created by a CREATE SCHEMA statement can be used as SQL authorization IDs.

## Naming the Schema

The name you enter after the AUTHORIZATION keyword in the CREATE SCHEMA statement is the name of the schema and is used as an authorization ID for SQL objects (tables, views, synonyms) owned by that schema. The schema name, or authorization ID (AUTHID), must be 1 to 18 characters in length. CA Datacom Datadictionary uses this SQL name as the name of the AUTHORIZATION occurrence which is defined to the CA Datacom Datadictionary. The name must be unique for all AUTHORIZATION occurrences. You cannot create a schema which has the same name as an existing AUTHORIZATION occurrence in CA Datacom Datadictionary.



## Relating the Person to the AUTHID

Your PERSON occurrence, identified to CA Datacom Datadictionary with the user signon ID, must be related to a valid SQL AUTHORIZATION occurrence using the PER-ATZ-AUTHID relationship before you can use the Interactive SQL Service Facility. This is a *many-to-one* relationship, meaning that several PERSON occurrences can be related to a specific AUTHORIZATION occurrence. Several users can have the same default AUTHID, but each user can have only one default.

When you access the Interactive SQL Service Facility, CA Datacom Datadictionary checks for the existence of the relationship between your PERSON occurrence and a valid SQL AUTHORIZATION occurrence. If a relationship does not exist, CA Datacom Datadictionary presents the following panel.

```

=>
=>
=>
-----
Interactive SQL Service Facility
                                     S01A
                Default Authorization Panel

Please enter your default authorization ID and press Enter.
AUTHID: _____

PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD   PF8=FORWARD
PF9=EXECUTE   PF10=LEFT     PF11=RIGHT     PF12=ALTERNATE

```

The AUTHORIZATION occurrence you specify on this panel becomes your permanent default AUTHID, until you change it. You can change the AUTHID either temporarily for the session or permanently. See [Changing Your AUTHID](#) (see page 402).

When the relationship exists, the Default Authorization Panel is not presented. The AUTHORIZATION occurrence is your default AUTHID and is displayed in the Current Authid field on the Source Member Panel.

## Changing Your AUTHID

Use the RESET DEFAULT AUTHID xxxxxxxxxxxxxxxxxxxx command, where the x's represent the 1 to 18 characters of an existing SQL AUTHORIZATION occurrence name, to change your default authorization ID. When the command completes successfully, the current PER-ATZ-AUTHID relationship is deleted and the new one is added.

Alternately, see the *CA Datacom Datadictionary Online Reference Guide* for information on deleting and adding relationships. You can use the online functions to change the PER-ATZ-AUTHID relationship and establish a new default authorization ID.

To change to your default AUTHID later in the session or to temporarily change the AUTHID, use the AUTHORIZATION-ID Panel. Obtain an AUTHORIZATION-ID Panel by either using the SET AUTHID command or by selecting the SET AUTHID function on the SQLADMIN Panel (see [Setting the Session Authorization ID](#) (see page 448)). When you have obtained an AUTHORIZATION-ID Panel, you can change to the default AUTHID by pressing PF9 (EXECUTE) without placing an entry on the panel. Or, if you want to temporarily change the AUTHID, you can reset it by placing an entry on the AUTHORIZATION-ID Panel and pressing PF9 (EXECUTE). In either case, after you press PF9 CA Datacom Datadictionary returns a message to the panel naming the authorization ID now in effect. Reset AUTHIDs remain in effect for the duration of the session or until again changed.

If you establish a session AUTHID and subsequently issue a RESET DEFAULT AUTHID command, the session AUTHID continues to be used during the current session.

## System Schemas

When upgrading from a version previous to CA Datacom/DB Release 8.0, all tables are assigned an AUTHID of SYSUSR. To access these tables, you must use the full SQL name. For example, a table with the SQL name of PAYROLL would be accessed by entering SYSUSR.PAYROLL.

**Important!** Do not relate any PERSON occurrence to the SYSUSR AUTHORIZATION occurrence with the Default Authorization Panel, the SET commands, or using the online facilities with the PER-ATZ-AUTHID relationship.

**Note:** When CA Datacom Datadictionary is initially installed, there are only two schemas: SYSADM for CA Datacom system tables and SYSUSR, as previously noted, for upgraded tables.

If the CA Datacom/DB Security Facility is installed, the SYSADM AUTHID can be used by a global database owner to create the first user schema after installation, but we recommend that you do not use SYSADM as an AUTHID thereafter. See the *CA Datacom Security Reference Guide* for information on global database owner status.

## Displaying and Reporting

You can display a table of AUTHORIZATION entity-occurrences using the CA Datacom Datadictionary Entity Display (ENTDISPL) Mode and view the attributes to determine if AUTH-USAGE=S. This indicates it is a valid SQL authorization ID. See the *CA Datacom Datadictionary Online Reference Guide* for information on the Entity Display Mode.

You can also run a batch report using the -RPT SCHEMA transaction to list all valid SQL authorization IDs and objects owned by each. See the *CA Datacom Datadictionary Batch Reference Guide* for information on the Schema Report.

## Example Source Member

The following panel shows an example of entering a CREATE SCHEMA statement. See [How to Submit SQL Statements](#) (see page 358) for the steps to obtain the Source Panel and an explanation of the fields.

```

=>
=>
=>

-----
Interactive SQL Service Facility                               SQLMAINT
                               Source Panel                      S01S
EDIT   Member: $DDSQL   Output Line Limit:  01000
       Person:  DATACOM-INSTALL
       Current Authid: SYSADM
       Description: CREATE A SCHEMA
-----
===== T O P =====
000001 create schema authorization jones;
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD  PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT    PF12=ALTERNATE

```

The statement CREATE SCHEMA and the keyword AUTHORIZATION are required. AUTHORIZATION must be followed by the name of the authorization ID you are creating. In this example, the authorization ID is JONES.

You can create a base table at the same time you create a schema by entering a CREATE TABLE statement in the source member.

You can create more than one schema in a source member by entering multiple CREATE SCHEMA statements. If you choose to do this, each authorization ID you specify must be unique, that is to say, no duplications are allowed, including naming an authorization ID which already exists.

For more information on the CREATE SCHEMA statement, see [CREATE SCHEMA](#) (see page 676).

After placing your SQL statement in the numbered line area on the Source Panel, you can perform the following:

- Enter additional SQL statements. Place a semicolon (;) after each complete statement.
- Press PF9 (EXECUTE) to execute the source member. CA Datacom Datadictionary responds with the Output Panel which shows the results of the SQL processing.
- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press the CLEAR key to return to the Interactive SQL Service Facility Panel where you can choose another option.

## Example Output Member

The following example is the Output Panel received after executing the previous CREATE SCHEMA statement.

```

=>
=>
=>

----- >>>
Interactive SQL Service Facility                               SQLMAINT

                                Output Panel                    S010
                                EDIT    Member= $DDSQL
                                Description: CREATE A SCHEMA
-----

===== T O P =====
000001 (OUTPUT CREATED FROM SOURCE MEMBER $DDSQL )
000002 create schema authorization jones;
000003 -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
000004 STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
000005 -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
000006 SQL STATEMENTS ASSUMED TO BE BETWEEN COLUMNS 1 AND 72
000007 NUMBER OF INPUT RECORDS READ IS 0001
000008 NUMBER OF SQL STATEMENTS PROCESSED IS 0001
===== B O T T O M =====
PF1=HELP      PF2=END      PF3=SPLIT    PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD PF8=FORWARD
PF9=EXECUTE   PF10=LEFT   PF11=RIGHT   PF12=ALTERNATE

```

To exit from the Output Panel, you can perform one of the following.

- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press PF12 (ALTERNATE) to display the Source Panel.
- Press the CLEAR key to return to the Interactive SQL Service Facility Panel.

## Creating a Table

Use a CREATE TABLE statement to define a table to CA Datacom/DB and CA Datacom Datadictionary. Enter the CREATE TABLE statement in the numbered area of an EDIT Source Member Panel. See [How to Submit SQL Statements](#) (see page 358) for the steps to obtain the panel and an explanation of the fields on the panel. See [CREATE TABLE](#) (see page 680) for the syntax of the statement. The SQL statement performs the following:

- Specify the table name.
- Define the columns in the table and specify their data type and length (the columns in the table are defined as FIELD occurrences in CA Datacom Datadictionary ). The columns and tables must follow the SQL naming conventions. See [Naming Conventions](#) (see page 480).

- Designate if one or more columns are to have a unique value for each row in the table. Use of the UNIQUE constraint generates a KEY entity-occurrence in the 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.

- Designate other constraints in addition to UNIQUE. See the sections starting with [Column Constraint Definition](#) (see page 684).
- Indicate the area where the table data is to reside.

## Naming the Table

The name you give the table in the CREATE TABLE statement is the SQL name you use when referencing the table in SQL statements.

The unqualified SQL table name is:

- 1 to 18 characters in length (to comply with ANSI or FIPS standards), or
- 1 to 32 characters in length (to take advantage of the CA Datacom/DB extended mode for SQL).

The qualified table name is the table name preceded by its schema and a period (.). The schema is represented by a valid authorization ID. Therefore, the qualified table name is in the format *authid.table-name*.

You can specify the authorization ID in the CREATE TABLE statement. If you do not specify an authorization ID, CA Datacom Datadictionary uses the current default authorization ID for the session.

The default authorization ID is the AUTHORIZATION occurrence related to the PERSON occurrence of the user ID entered to sign on to CA Datacom Datadictionary. However, you can establish an authorization ID that is the default for the current session. See [Relating the Person to the AUTHID](#) (see page 401).

CA Datacom Datadictionary uses the authorization ID and the table's SQL name to build the TABLE entity-occurrence name, as follows:

- The period (.) separating the AUTHID and the SQL name is replaced with a dash (-).
- If the combination of the authorization ID, the SQL name and the dash is greater than the maximum allowed length of 32 characters, CA Datacom Datadictionary truncates the concatenated name.
- If the truncated name already exists, CA Datacom Datadictionary truncates it by four more characters and adds four zeros to the end of the name. CA Datacom Datadictionary increments the digits by 1 until the name is unique.

You can change the table's CA Datacom Datadictionary entity-occurrence name by restoring it to TEST status and using one of the following CA Datacom Datadictionary rename functions:

- The UPDATE NAME option of the CA Datacom Datadictionary Entity Maintenance (ENTMAINT) Mode.
- The NAM or NME margin commands in the CA Datacom Datadictionary CA Datacom/DB Structure Maintenance (DBMAINT) Mode.
- The 1000 NEWNAME batch transaction.

If you rename the TABLE entity-occurrence, you must recatalog it to the CA Datacom/DB Directory (CXX).

**Note:** Before copying back to PRODUCTION status, drop the table that currently exists there.

See the *CA Datacom Datadictionary Online Reference Guide* and *CA Datacom Datadictionary Batch Reference Guide* for more information on renaming entity-occurrences.

## Key Creation

You can create CA Datacom Datadictionary KEY entity-occurrences in several ways with parameters in your SQL statements and by using the CREATE INDEX statement. See [CREATE TABLE](#) (see page 680) and [Creating an Index](#) (see page 415) for details. For information about specifying an SQL key selection override key in either the correlation name or synonym name, see [Overriding SQL Key Selection](#) (see page 143).

The following are four of the ways KEY entity-occurrences can be defined by CA Datacom Datadictionary as a result of SQL processing. The key SQL name is created by CA Datacom Datadictionary. The name consists of the DATACOM-NAME attribute-value of the key followed by the underscore character followed by the concatenation of the DATACOM-NAME attribute-value of the table and the DATACOM-ID attribute-value of the database containing the key. For example, a key with DATACOM-NAME SQ032 in table INV in database 16 has the SQLNAME attribute-value SQ032\_INV00016.

- The first column you specify in the CREATE TABLE statement becomes the CA Datacom/DB Master and Native Key for the table except when a primary or unique key has been defined. CA Datacom Datadictionary automatically generates this KEY entity-occurrence and gives it the same name as the CA Datacom/DB five-character name it generates (SQnnn) and SQLNAME=SQLKEY.

**Note:** The DUPE-MASTER-KEY and CHNG-MASTER-KEY attributes of the TABLE entity-occurrence are set to Y, indicating that the value of the Master Key can be duplicated and/or changed.

- A foreign key can be generated. Its name is the same as the constraint name. A physical key will not be generated in the Directory.
- If you use the column-level UNIQUE constraint (after an individual column name), a KEY entity-occurrence is generated. The CA Datacom Datadictionary name for the key is the same as the CA Datacom/DB five-character name it generates (SQnnn).  
**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.
- If you use the table-level UNIQUE constraint or primary key (followed by a list of one or more column names enclosed in parentheses), a KEY entity-occurrence is generated. The CA Datacom Datadictionary name of the key is the same as the CA Datacom/DB five-character name it generates (SQnnn).  
**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.

The five-character DATACOM name of each key uses SQ followed by three digits for uniqueness.

## Element Creation

CA Datacom Datadictionary creates an element that includes all the columns in the table. The CA Datacom Datadictionary name of the element is the same as the table and the five-character CA Datacom name of the element is SQLEL. You can rename the element, change the element, and add more elements to the table using the other modes of CA Datacom Datadictionary.



## Statement Execution Results

When the statement executes successfully, the table, columns, key, and element are defined to CA Datacom Datadictionary in PROD status and cataloged to the CA Datacom/DB Directory (CXX).

You can use the other modes of CA Datacom Datadictionary to add support data, such as aliases, descriptors, additional relationship definitions, and text. You can rename keys, change keys, and add more keys to the table using the other modes of CA Datacom Datadictionary.

To add additional keys or elements, copy the PROD status version of the table to a TEST status, make your modifications, and catalog the table using the CA Datacom Datadictionary CATALOG function.

**Note:** For information on enhancing the table definitions, see the *CA Datacom/DB Database and System Administration Guide* and the *CA Datacom Datadictionary User Guide*.

## Example Source Member

The following example shows a CREATE TABLE statement. See [How to Submit SQL Statements](#) (see page 358) for the steps to obtain the Source Panel and an explanation of the fields.

```

=>
=>
=>

-----
Interactive SQL Service Facility                               SQLMAINT
                                     Source Panel                S01S
      EDIT      Member: $DDSQL   Output Line Limit: 01000
                Person: JONES
                Current Authid: JONES
                Description: CREATE TABLE DEPTTBL
-----
===== T O P =====
000001 create table depttbl
..... (deptno char(2) not null,
..... deptname char(24) not null,
..... mgrnbr char(6) not null,
..... unique (deptno, mgrnbr));
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD PF8=FORWARD
PF9=EXECUTE   PF10=LEFT     PF11=RIGHT    PF12=ALTERNATE

```

Before entering the statement, we inserted four additional lines on the panel by typing *i4* in the line number and pressing Enter.

The statement CREATE TABLE is required and must be followed by the name of the table, which is DEPTTBL in this example. Since the name of a data area in which the table is to reside is not specified, it is placed in the default area.

Each column definition is listed on a separate line and indented from the left simply for ease of reading. In this example, all the columns are of the CHARACTER data type. The short form, CHAR, is used, followed by the length in parentheses. NOT NULL is specified for the first three columns so that these columns cannot contain null values. The list of column names must be enclosed in parentheses. You must separate the column definitions from each other with a comma.

The table-level UNIQUE constraint specifies that the combined values of the DEPTNO and MGRNBR columns are to be unique for each row of this table. The columns specified in a table-level UNIQUE constraint must be defined with NOT NULL. The column names must be separated by commas and the list enclosed in parentheses when using this form of the UNIQUE constraint.

For information on the syntax of the CREATE TABLE statement, see [CREATE TABLE](#) (see page 680).

After placing your SQL statement in the numbered line area on the Source Panel, you can perform the following:

- Enter additional SQL statements. Place a semicolon (;) after each complete statement.
- Press PF9 (EXECUTE) to execute the source member. CA Datacom Datadictionary responds with the Output Panel which shows the results of the SQL processing.
- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press the CLEAR key to return to the Interactive SQL Service Facility Panel where you can choose another option.

## Example Output Member

The following example is the Output Panel received when executing the previous CREATE TABLE statement.

```

=>
=>
=>

----- >>>
Interactive SQL Service Facility                               SQLMAINT

                                Output Panel                    S010
                                EDIT    Member= DEPTTBL
                                Description: CREATE TABLE DEPTTBL
-----

===== T O P =====
000001 (OUTPUT CREATED FROM SOURCE MEMBER DEPTTBL )
000002 create table depttbl
000003 (deptno char(2) not null,
000004 deptname char(24) not null,
000005 mgrnbr char(6) not null,
000006 unique (deptno, mgrnbr));
000007 -----+-----+-----+-----+-----+-----+-----+
000008 STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
000009 -----+-----+-----+-----+-----+-----+-----+
PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS      MORE...
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD   PF8=FORWARD
PF9=EXECUTE   PF10=LEFT     PF11=RIGHT     PF12=ALTERNATE

```

If the display is larger than your screen allows, use the PF keys to scroll to other portions of the display.

```

=>
=>
=>

----- >>>
Interactive SQL Service Facility                               SQLMAINT

                                Output Panel                    S010
                                EDIT    Member= DEPTTBL
                                Description: CREATE TABLE DEPTTBL
-----

000009 -----+-----+-----+-----+-----+-----+-----+
000010 SQL STATEMENTS ASSUMED TO BE BETWEEN COLUMNS 1 AND 72
000011 NUMBER OF INPUT RECORDS READ IS 0006
000012 NUMBER OF SQL STATEMENTS PROCESSED IS 0001
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD   PF8=FORWARD
PF9=EXECUTE   PF10=LEFT     PF11=RIGHT     PF12=ALTERNATE

```

In this example, a table is created with the full SQL name of JONES.DEPTTBL. The columns in the table are all character data type with lengths as specified. Since an area is not specified, the table is placed in the default area.

The combined values for the DEPTNO and MGRNBR columns must be unique for each row in the table.

Two KEY entity-occurrences are generated by CA Datacom Datadictionary during execution of the CREATE TABLE statement. The CA Datacom/DB Master and Native Key is the first column of the table, DEPTNO. The table-level UNIQUE constraint causes the generation of another KEY entity-occurrence, which contains both the DEPTNO and MGRNBR columns.

To exit from the Output Panel, you can perform one of the following.

- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press PF12 (ALTERNATE) to display the Source Panel.
- Press the CLEAR key to return to the Interactive SQL Service Facility Panel.

## Altering a Table

Use the ALTER TABLE statement to change a table definition without manually reloading it. You can change one of the table's column definitions and/or the table's constraints in a single statement. You can make only one change to a column in the ALTER TABLE statement.

You can perform the following with this statement:

- Define a new column. (The columns in the table are defined as FIELD occurrences in CA Datacom Datadictionary.)
- Define new table constraints.
- Drop constraints, columns, foreign keys or primary keys. (see the sections starting with Column Constraint Definition.)
- Modify a column definition.
- Rename a column.

Enter the ALTER TABLE statement in the numbered area of an EDIT Source Member Panel. See [How to Submit SQL Statements](#) (see page 358) for the steps to obtain the panel and an explanation of the fields on the panel. See [ALTER TABLE](#) (see page 598) for information and the syntax of the statement.

## Statement Execution Results

When the statement executes successfully, the changes are defined to CA Datacom Datadictionary in PRODUCTION status and cataloged to the CA Datacom/DB Directory (CXX).

You can use the other modes of CA Datacom Datadictionary to add support data, such as aliases, descriptors, additional relationship definitions, and text. For information on enhancing the table definitions, see the *CA Datacom/DB Database and System Administration Guide* and the *CA Datacom Datadictionary User Guide*.

If the ALTER TABLE statement cannot complete, it is rolled back to its beginning. If the ALTER TABLE process is force checkpointed (because of the size of the Log Area (LXX) and the concurrent activity), the rollback is incomplete. Therefore, before altering large tables ensure that the Log Area (LXX) is large enough and take a backup of the data area.

## Example Source Member

The following example shows an ALTER TABLE statement. See [How to Submit SQL Statements](#) (see page 358) for the steps to obtain the Source Panel and an explanation of the fields.

```

=>
=>
=>

-----
Interactive SQL Service Facility                               SQLMAINT
                                     Source Panel                S01S
EDIT      Member: DEPTTBL2  Output Line Limit: 01000
          Person: JONES
          Current Authid: JONES
          Description: ADD ADMDEPT TO DEPTTBL
-----
===== T O P =====
000001 alter table depttbl
.....  add admdept char(2);
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT    PF12=ALTERNATE

```

Before entering the statement, we inserted an additional line on the panel by typing *i1* in the line number and pressing Enter.

The statement ALTER TABLE is required and must be followed by the name of the table, which is DEPTTBL in this example.

The column definition is listed on a separate line. Since only one column is added, the parentheses are not required.

For information on the ALTER TABLE statement, see [ALTER TABLE](#) (see page 598).

After placing your SQL statement in the numbered line area on the Source Panel, you can perform the following:

- Enter additional SQL statements. Place a semicolon (;) after each complete statement.
- Press PF9 (EXECUTE) to execute the source member. CA Datacom Datadictionary responds with the Output Panel which shows the results of the SQL processing.
- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press the CLEAR key to return to the Interactive SQL Service Facility Panel where you can choose another option.

### Example Output Member

The following example is the Output Panel received when executing the previous statement.

```

=>
=>
=>

----- >>>
Interactive SQL Service Facility                               SQLMAINT
                    Output Panel                               S010
          EDIT      Member= DEPTTBL2
          Description: ADD ADMDEPT TO DEPTTBL
-----
===== T O P =====
000001 (OUTPUT CREATED FROM SOURCE MEMBER DEPTTBL2)
000002 alter table depttbl
000003   add admdept char(2);
000004 -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
000005 STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
000006 -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
000007 SQL STATEMENTS ASSUMED TO BE BETWEEN COLUMNS 1 AND 72
000008 NUMBER OF INPUT RECORDS READ IS 0002
000009 NUMBER OF SQL STATEMENTS PROCESSED IS 0001
PF1=HELP       PF2=END           PF3=SPLIT      PF4=PROCESS    MORE...
PF5=TOP        PF6=BOTTOM        PF7=BACKWARD   PF8=FORWARD
PF9=EXECUTE    PF10=LEFT         PF11=RIGHT     PF12=ALTERNATE
  
```

The column ADMDEPT is added after the existing columns in the table. If the table has existing rows, the rows receive NULL values in the added column unless NOT NULL WITH DEFAULT is specified with a column constraint definition. The length of the column is added to the table's element SQLEL.

To exit from the Output Panel, you can perform one of the following.

- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press PF12 (ALTERNATE) to display the Source Panel.
- Press the CLEAR key to return to the Interactive SQL Service Facility Panel.

## Creating an Index

Use a CREATE INDEX statement to define a key to CA Datacom/DB and the CA Datacom Datadictionary. When a CREATE INDEX statement successfully executes, a KEY entity-occurrence is defined in CA Datacom Datadictionary in PROD status. The five-character DATACOM name of the key is generated for you by CA Datacom Datadictionary. The key's attributes include MASTER-KEY=N, NATIVE-KEY=N, INCLUDE-NIL-KEY=Y, and UNIQUE=N. For information about specifying an SQL key selection override key in either the correlation name or synonym name, see [Overriding SQL Key Selection](#).

CREATE INDEX causes all plans dependent on the indexed table to be 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 on running a Relationship Report.

Enter the CREATE INDEX statement in the numbered area of an EDIT Source Member Panel. See [How to Submit SQL Statements](#) (see page 358) for the steps to obtain the panel and an explanation of the fields on the panel. See [CREATE INDEX](#) (see page 618) for the syntax of the statement.

The SQL statement performs the following:

- Specify the index name.
- Specify the table to which the index belongs.
- Define the columns in the index.

## Naming the Index (Key)

The name you place in the CREATE INDEX statement is the SQL name you use when referencing the key in SQL statements.

The unqualified SQL name is:

- 1 to 18 characters in length to comply with ANSI or FIPS standards.
- 1 to 32 characters in length to take advantage of the CA Datacom/DB extended mode for SQL.

The qualified index name is the index name preceded by its schema and a period (.). The schema is represented by a valid authorization ID. Therefore, the qualified index name is in the format *authid.index-name*.

You can specify the authorization ID in the CREATE INDEX statement. If you do not specify an authorization ID, CA Datacom Datadictionary uses the current default authorization ID for the session.

The default authorization ID is the AUTHORIZATION occurrence related to the PERSON occurrence of the user ID entered to sign on to CA Datacom Datadictionary. However, you can establish an authorization ID that is the default for the current session. See [Relating the Person to the AUTHID](#) (see page 401).

## Key Creation

CREATE INDEX generates a KEY entity-occurrence. See [CREATE INDEX](#) (see page 618) for details.

The five-character DATACOM name of each key uses SQ followed by three digits for uniqueness.

## Statement Execution Results

When the statement executes successfully, a KEY entity-occurrence is defined to CA Datacom Datadictionary in PROD status and cataloged to the CA Datacom/DB Directory (CXX).

You can use the other modes of CA Datacom Datadictionary to add support data, such as aliases, descriptors, additional relationship definitions, and text. You can rename keys, change keys, and add more keys to the table using the other modes of CA Datacom Datadictionary.



## Example Source Member

The following example shows a CREATE INDEX statement. See [How to Submit SQL Statements](#) (see page 358) for the steps to obtain the Source Panel and an explanation of the fields.

```

=>
=>
=>

-----
Interactive SQL Service Facility                                SQLMAINT
                                Source Panel                                S01S
EDIT   Member: $DDSQL   Output Line Limit: 01000
        Person: JONES
        Current Authid: JONES
        Description: CREATE INDEX EMPLOYEE_INDEX
-----
===== T O P =====
000001 create index employee_index on employees (empno);
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD PF8=FORWARD
PF9=EXECUTE   PF10=LEFT   PF11=RIGHT    PF12=ALTERNATE

```

The statement CREATE INDEX is required and must be followed by the name of the index (employee\_index in this example) the word "on", the table name (employees), and the column-list (empno). For information on the syntax of the CREATE INDEX statement, see [CREATE INDEX](#) (see page 618).

After placing your SQL statement in the numbered line area on the Source Panel, you can perform the following:

- Enter additional SQL statements. Place a semicolon (;) after each complete statement.
- Press PF9 (EXECUTE) to execute the source member. CA Datacom Datadictionary responds with the Output Panel which shows the results of the SQL processing.
- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press the CLEAR key to return to the Interactive SQL Service Facility Panel where you can choose another option.



---

## Creating a View

Use the CREATE VIEW statement to define a view (a derived table) of one or more tables or views. Enter the CREATE VIEW statement in the numbered area of an EDIT Source Member Panel. See [How to Submit SQL Statements](#) for the steps to obtain the panel and an explanation of the fields on the panel. See [CREATE VIEW](#) (see page 705) for the syntax of the statement.

The following restrictions apply to the tables on which a view is derived.

### SQL Accessible Tables

Since a view is derived from specified columns in a table or tables, the table or tables must be defined in the CA Datacom Datadictionary and be SQL accessible. You can place the CREATE VIEW statement(s) after the the CREATE TABLE statement(s) in a single source member.

### Same Security Type

All of the tables and views specified with the CREATE VIEW statement must be in databases secured under the same security model, either the CA Datacom/DB External Security Model or the SQL Security Model. See the *CA Datacom Security Reference Guide* for more information about security models.

### Remote Tables

A complete duplicate definition of a remote table in CA Datacom Datadictionary is required for SQL access. No version control enforcement is available to ensure that remote definitions are synchronized with the local active definitions.

## Naming the View

The name you give the view in the CREATE VIEW statement is the SQL name you use when referencing the view in SQL statements.

The unqualified view name is:

- 1 to 18 characters in length if you want it to comply with ANSI or FIPS standards.
- 1 to 32 characters in length if you want to take advantage of the CA Datacom/DB extended mode for SQL.

The qualified view name is the view name preceded by its schema and a period (.). The schema is represented by a valid authorization ID. Therefore, the qualified view name is in the format *authid.view-name*.

You can specify the authorization ID in the CREATE VIEW statement. If you do not specify an authorization ID, CA Datacom Datadictionary uses the current default authorization ID for the session.

The default authorization ID is the AUTHORIZATION occurrence related to the PERSON occurrence of the user ID entered to sign on to CA Datacom Datadictionary. However, you can establish an authorization ID that is the default for the current session. See [Relating the Person to the AUTHID](#) (see page 401).

CA Datacom Datadictionary uses the authorization ID and the view's SQL name to build the VIEW entity-occurrence name, as follows:

- The period (.) separating the AUTHID and the SQL name is replaced with a dash (-).
- If the combination of the authorization ID, the SQL name and the dash is greater than the maximum allowed length of 32 characters, CA Datacom Datadictionary truncates the concatenated name.
- If the truncated name already exists, CA Datacom Datadictionary truncates it by four more characters and adds four zeros to the end of the name. CA Datacom Datadictionary increments the digits by 1 until the name is unique.

### Example Source Member

The following example shows the CREATE VIEW statement. See [How to Submit SQL Statements](#) (see page 358) for the steps to obtain the Source Panel and an explanation of the fields.

```

=>
=>
=>

-----
Interactive SQL Service Facility                               SQLMAINT
                                     Source Panel                S01S
      EDIT      Member: DEPTVIEW  Output Line Limit:  01000
                Person: JONES
      Current Authid: JONES
                Description: VIEW DEPARTMENT ORGANIZATION
-----
===== T O P =====
000001 create view deptview
.....      (deptno, deptname, admdept)
..... as select all
.....      deptno, deptname, admdept
..... from depttbl
..... where admdept = 'A1';
===== B O T T O M =====
PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD  PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT    PF12=ALTERNATE

```

Before entering the statement, we inserted five additional lines on the panel by typing *i5* in the line number and pressing Enter.

The statement CREATE VIEW is required and must be followed by the name of the view, which is DEPTVIEW in this example.

The names of the columns in the view are listed on one line in this example. The column names must be separated by commas and enclosed in parentheses. Since the view is based on an existing table, the view columns have the same data type as the table column. The first column named for the view corresponds to the first column named in the SELECT statement, the second column for the view corresponds to the second column named in the SELECT, and so on. In the previous example, the view columns have the same names as the table columns.

The WHERE clause limits the rows in this view to those rows in the table where the value of the ADMDEPT column is equal to the literal value 'A1'.

For more information on the syntax of the CREATE VIEW statement, see [CREATE VIEW](#) (see page 705).

After placing your SQL statement in the numbered line area on the Source Panel, you can perform the following:

- Enter additional SQL statements. Place a semicolon (;) after each complete statement.
- Press PF9 (EXECUTE) to execute the source member. CA Datacom Datadictionary responds with the Output Panel which shows the results of the SQL processing.
- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press the CLEAR key to return to the Interactive SQL Service Facility Panel where you can choose another option.

## Example Output Member

The following example shows the Output Panel received when executing the previous CREATE VIEW statement.

```

=>
=>
=>

----- >>>
Interactive SQL Service Facility                               SQLMAINT

                                Output Panel                    S010
                                EDIT      Member= DEPTVIEW
                                Description: VIEW DEPARTMENT ORGANIZATION
-----

===== T O P =====
000001 (OUTPUT CREATED FROM SOURCE MEMBER DEPTVIEW )
000002 create view deptview
000003      (deptno, deptname, admdept)
000004 as select all
000005      deptno, deptname, admdept
000006 from depttbl
000007 -----+-----+-----+-----+-----+-----+-----+-----+
000008 where admdept = 'A1';
000009 -----+-----+-----+-----+-----+-----+-----+
PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS      MORE...
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD   PF8=FORWARD
PF9=EXECUTE   PF10=LEFT     PF11=RIGHT     PF12=ALTERNATE

```

If the display is larger than your screen allows, use the PF keys to scroll to other portions of the display.

```

=>
=>
=>

----- >>>
Interactive SQL Service Facility                               SQLMAINT

                                Output Panel                    S010
                                EDIT      Member= DEPTVIEW
                                Description: VIEW DEPARTMENT ORGANIZATION
-----

000009 -----+-----+-----+-----+-----+-----+-----+
000010 STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
000011 -----+-----+-----+-----+-----+-----+-----+
000012 SQL STATEMENTS ASSUMED TO BE BETWEEN COLUMNS 1 AND 72
000013 NUMBER OF INPUT RECORDS READ IS 0006
000014 NUMBER OF SQL STATEMENTS PROCESSED IS 0001
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD   PF8=FORWARD
PF9=EXECUTE   PF10=LEFT     PF11=RIGHT     PF12=ALTERNATE

```

The previous example of a view is a row and column subset view of the table DEPTTBL which contains columns DEPTNO, DEPTNAME, MGRNBR, and ADMDEPT. The view DEPTVEW is a subset since only three of the four columns in DEPTTBL are included in the view, plus the number of rows is limited to those where the value of the ADMDEPT column is equal to the literal value 'A1'.

This view can be used to update values for the three specified columns. However, this view would not be used to insert new rows into the DEPTTBL table since one of the columns is not included in the view definition.

To exit from the Output Panel, you can perform one of the following:

- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press PF12 (ALTERNATE) to display the Source Panel.
- Press the CLEAR key to return to the Interactive SQL Service Facility Panel.

## Creating a Synonym

Use the CREATE SYNONYM statement to define an alternative name for a table or view. However, the table or view must be defined in CA Datacom Datadictionary before you can create its synonym. After defining a synonym, you can use it in place of the full qualified table or view name (authorization ID and table or view name).

You can define synonyms in the same schema as the tables or views. However, the schema must be represented by the current authorization ID for the session. You can define a synonym in your schema for a table or view that is listed in another schema.

Enter the CREATE SYNONYM statement in the numbered area of an EDIT Source Member Panel. See [How to Submit SQL Statements](#) (see page 358) for the steps to obtain the panel and an explanation of the fields on the panel. See [CREATE SYNONYM](#) (see page 678) for the syntax of the statement.

## Naming the Synonym

The name you give the synonym in the CREATE SYNONYM statement is the SQL name you use when referencing the synonym in SQL statements.

The SQL synonym name is:

- 1 to 18 characters in length if you want it to comply with ANSI or FIPS standards.
- 1 to 32 characters in length if you want to take advantage of the CA Datacom/DB extended mode for SQL.

The CREATE SYNONYM statement syntax does not allow you to specify an authorization ID before the synonym name. Since you cannot specify an authorization ID, CA Datacom Datadictionary uses the current default authorization ID.

The default authorization ID is the AUTHORIZATION occurrence related to the PERSON occurrence of the user ID entered to sign on to CA Datacom Datadictionary or the current authorization ID established with the SET AUTHID function. See [Relating the Person to the AUTHID](#) (see page 401).

Since you cannot prefix the synonym name with an authorization ID in the CREATE SYNONYM statement, CA Datacom Datadictionary automatically prefixes the synonym name with the current authorization ID for the online session. If the combined authorization ID and synonym name is greater than the maximum allowed length of 32 characters, CA Datacom Datadictionary truncates the concatenated name.

If the truncated name already exists, CA Datacom Datadictionary truncates it by four more characters and adds four zeros to the end of the name. CA Datacom Datadictionary increments the digits by 1 until the name is unique.



## Example Source Member

The following example shows the CREATE SYNONYM statement.

```

=>
=>
=>

-----
Interactive SQL Service Facility                                SQLMAINT
                                     Source Panel                                S01S
EDIT   Member: PROJSYN   Output Line Limit: 01000
        Person: JONES
        Current Authid: JONES
        Description: CREATE SYNONYM PROJECTTBL
-----
===== T O P =====
000001 create synonym projects for ted.projecttbl;
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD   PF8=FORWARD
PF9=EXECUTE   PF10=LEFT     PF11=RIGHT     PF12=ALTERNATE

```

The statement CREATE SYNONYM is required and must be followed by the name of the synonym, which is PROJECTS in this example.

The FOR clause specifies the authorization ID of the table's schema and the name of the table. The authorization ID and table name are concatenated by the period (.) indicating this is the qualified name of the table. The table name must be qualified in this case since it is not in the schema identified by the authorization ID (JONES) which is the default authorization ID.

For more information on the syntax of the CREATE SYNONYM statement, see [CREATE SYNONYM](#). (see page 678)

After placing your SQL statement in the numbered line area on the Source Panel, you can perform the following:

- Enter additional SQL statements. Place a semicolon (;) after each complete statement.
- Press PF9 (EXECUTE) to execute the source member. CA Datacom Datadictionary responds with the Output Panel which shows the results of the SQL processing.
- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press the CLEAR key to return to the Interactive SQL Service Facility Panel where you can choose another option.

## Example Output Member

The following example shows the Output Panel received when executing the previous CREATE SYNONYM statement.

```

=>
=>
=>

----->>>
Interactive SQL Service Facility                                SQLMAINT

                                Output Panel                    S010
                                EDIT    Member= PROJSYN
                                Description: CREATE SYNONYM PROJECTTBL
-----
===== T O P =====
000001 (OUTPUT CREATED FROM SOURCE MEMBER PROJSYN )
000002 create synonym projects for ted.projecttbl;
000003 -----+-----+-----+-----+-----+-----+-----+
000004 STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
000005 -----+-----+-----+-----+-----+-----+-----+
000006 SQL STATEMENTS ASSUMED TO BE BETWEEN COLUMNS 1 AND 72
000007 NUMBER OF INPUT RECORDS READ IS 0001
000008 NUMBER OF SQL STATEMENTS PROCESSED IS 0001
===== B O T T O M =====
PF1=HELP          PF2=END          PF3=SPLIT          PF4=PROCESS
PF5=TOP           PF6=BOTTOM       PF7=BACKWARD       PF8=FORWARD
PF9=EXECUTE       PF10=LEFT        PF11=RIGHT         PF12=ALTERNATE

```

This example of creating a synonym defines an alternative name, PROJECTS, for the PROJECTTBL table listed in the schema that has the authorization ID of TED. The synonym is listed in the schema with the authorization ID of JONES.

To exit from the Output Panel, you can perform one of the following:

- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press PF12 (ALTERNATE) to display the Source Panel.
- Press the CLEAR key to return to the Interactive SQL Service Facility Panel.

## Adding and Replacing Comments

You can use the `COMMENT ON` statement to provide descriptive information for a table, view or column. You can place the statement after the `CREATE` statement on the same Source Panel or in its own Source Panel. CA Datacom Datadictionary places the comment for the object in a text classification named `COMMENT`. When you add a comment to an object that already has a comment, the new comment replaces the old comment.

You cannot retrieve a comment through SQL. You can display the text created with the `COMMENT ON` statement through the CA Datacom Datadictionary online or batch facilities. You can also add additional classifications of text to the occurrence using the CA Datacom Datadictionary functions. See the *CA Datacom Datadictionary User Guide* for more information on text classifications.

Enter the `COMMENT ON` statement in the numbered area of an EDIT Source Member Panel. See [How to Submit SQL Statements](#) (see page 358) for the steps to obtain the panel and an explanation of the fields on the panel. See [COMMENT ON](#) (see page 613) for the syntax of the statement.

### Example Source Member

The following example shows a `COMMENT ON` statement added to an existing table:

```

=>
=>
=>

-----
Interactive SQL Service Facility                               SQLMAINT
                                     Source Panel                S01S
EDIT      Member: DEPTCMNT  Output Line Limit: 01000
          Person: JONES
          Current Authid: JONES
          Description: DEPTTBL COMMENT
-----
===== T O P =====
000001 comment on table depttbl
..... is 'reflects 1st qtr 99 reorganization';
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD PF8=FORWARD
PF9=EXECUTE   PF10=LEFT   PF11=RIGHT    PF12=ALTERNATE

```

The statement COMMENT ON is required. In this example, TABLE is also required since the comment is being added for a table. (If the comment is for a view, the keyword VIEW is required, and if the comment is for a column, the keyword COLUMN is required.) The TABLE keyword is followed by the name of the table. The required keyword IS introduces the comment, which must be enclosed in apostrophes since it is a literal string.

The COMMENT ON statement also allows you to add comments for more than one column in a table. For more information on this special syntax and the COMMENT ON statement in general, see [COMMENT ON](#) (see page 613).

After placing your SQL statement in the numbered line area on the Source Panel, you can perform the following:

- Enter additional SQL statements. Place a semicolon (;) after each complete statement.
- Press PF9 (EXECUTE) to execute the source member. CA Datacom Datadictionary responds with the Output Panel which shows the results of the SQL processing.
- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press the CLEAR key to return to the Interactive SQL Service Facility Panel where you can choose another option.

## Example Output Member

When you execute the command, you receive a Output Panel similar to the following example.

```

=>
=>
=>

----- >>>
Interactive SQL Service Facility                               SQLMAINT

                    Output Panel                               S010
          EDIT      Member= DEPTCMNT
                Description: DEPTTBL COMMENT
-----

===== T O P =====
000001 (OUTPUT CREATED FROM SOURCE MEMBER DEPTCMNT)
000002 comment on table depttbl
000003 is 'reflects 1st qtr 99 reorganization';
000004 -----+-----+-----+-----+-----+-----+-----+-----+-----+
000005 STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
000006 -----+-----+-----+-----+-----+-----+-----+-----+-----+
000007 SQL STATEMENTS ASSUMED TO BE BETWEEN COLUMNS 1 AND 72
000008 NUMBER OF INPUT RECORDS READ IS 0002
000009 NUMBER OF SQL STATEMENTS PROCESSED IS 0001
PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS      MORE...
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD   PF8=FORWARD
PF9=EXECUTE   PF10=LEFT     PF11=RIGHT     PF12=ALTERNATE

```

This is an example of a comment on table DEPTTBL. The comment is entered in the text classification of COMMENT for the occurrence definition in CA Datacom Datadictionary. See the *CA Datacom Datadictionary Online Reference Guide* for information on displaying text for an occurrence or the *CA Datacom Datadictionary Batch Reference Guide* for instructions on running a Text Report.

To exit from the Output Panel, you can perform one of the following:

- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press PF12 (ALTERNATE) to display the Source Panel.
- Press the CLEAR key to return to the Interactive SQL Service Facility Panel.



# Chapter 18: Deleting SQL Objects

---

You can use the DROP statement to completely remove tables, indexes, views, and synonyms. This function obsoletes all definitions of the object in the CA Datacom Datadictionary, deletes it from the CA Datacom/DB Directory, and removes it from the schema. You cannot use the DROP statement to remove a schema. See [Deleting a Schema](#) (see page 432).

Enter the DROP statement in the numbered area of an EDIT Source Member Panel. See [How to Submit SQL Statements](#) (see page 358) for the steps to obtain the panel and an explanation of the fields on the panel. See [DROP](#) (see page 725) for the syntax of the statement.

All application plans that reference the dropped object are marked invalid. In addition, any data stored in a table is deleted when it is dropped. 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 on running a Relationship Report.

- When you drop an index, all versions of the index in all statuses are automatically obsoleted.
- When you drop a table, all versions of the table in all statuses and all views and synonyms dependent on the table are automatically obsoleted.
- When you drop a view, all views and synonyms dependent on the view are automatically obsoleted.
- When you drop a synonym, only the synonym is obsoleted.

**Note:** The DROP statement is not processed, and you receive a -118 return code when the entity-occurrence definition of the index, table, view, or synonym you specify is protected with a password or a Lock Level 1 or 2 in CA Datacom Datadictionary. The error message also includes a CA Datacom Datadictionary Service Facility (DSF) return code. The DSF return codes are:

- IPW for password protected
- IOR for Lock Level 1 protected
- NTF for Lock Level 2 protected

See [Preliminary Information—Lock Levels](#) (see page 597) and to CA Datacom Datadictionary documentation for more information on passwords and lock levels.

For information about the CA Datacom/DB implementation of DROP PROCEDURE and DROP TRIGGER/RULE, see [Procedures and Triggers](#) (see page 70) and [Datadictionary Support for Triggers and Procedures](#) (see page 86).

## Deleting a Schema

You cannot use DROP to delete a schema. You must use the online CA Datacom Datadictionary Entity Maintenance (ENTMAINT) Mode to delete the authorization entity-occurrence that identifies the schema.

See the *CA Datacom Datadictionary Online Reference Guide* for information on using ENTMAINT Mode to delete an occurrence.

Before you can delete an SQL AUTHORIZATION occurrence, you must make certain it is not related to any existing occurrences (TABLE, VIEW, SYNONYM, PLAN) which use it as an authorization ID.

You can run a batch report to determine if the AUTHORIZATION occurrence is related to any other occurrences. See the *CA Datacom Datadictionary Batch Reference Guide* for information on running a Schema Report.

## Dropping an Index

The DROP INDEX statement deletes an index.

Make certain that the name you use in the DROP INDEX statement matches the SQLNAME attribute of the KEY that you wish to DROP.

Enter the DROP INDEX statement in the numbered area of an EDIT Source Member Panel. See [How to Submit SQL Statements](#) (see page 358) for the steps to obtain the panel and an explanation of the fields on the panel.

### **Specifying Authorization ID**

To be certain you are dropping a desired index, qualify the index name with the authorization ID in the DROP statement. If you specify an authorization ID for the index name, it must be the same as the authorization ID of the table name you specify in the FROM table-name clause in the DROP statement. If you do not specify the authorization ID, CA Datacom Datadictionary uses the current default authorization ID for the online session.

### **Impact**

Dropping an index removes the index from the Index Area (IXX) and removes the index definition from the Directory (CXX) and CA Datacom Datadictionary. 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 on running a Relationship Report.



### Example Source Member

The following example shows the DROP INDEX statement. See How to Submit SQL Statements for the steps to obtain the panel and an explanation of the fields on the panel.

```

=>
=>
=>

-----
Interactive SQL Service Facility                                SQLMAINT
                                     Source Panel                                S01S
      EDIT      Member: DINDEX      Output Line Limit: 01000
                Person: JONES
                Current Authid: JONES
                Description: DROP INDEX EMPLOYEE_INDEX
-----
===== T O P =====
000001 drop index employee_index from employees;
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD  PF8=FORWARD
PF9=EXECUTE   PF10=LEFT   PF11=RIGHT    PF12=ALTERNATE

```

The DROP statement is required, as are the keywords INDEX and FROM. The keyword INDEX must be followed by the name of the index (employee\_index in this example) you are dropping. The keyword FROM must be followed by the name of the table (employees in this example) to which the index belongs.

Once this source member is executed, you cannot recall the index definition. If you want to use this index again, you must re-create it.

See [DROP](#) (see page 725) for more information on the syntax of the DROP statement.

After placing your SQL statement in the numbered line area on the Source Panel, you can perform the following:

- Enter additional SQL statements. Place a semicolon (;) after each complete statement.
- Press PF9 (EXECUTE) to execute the source member. CA Datacom Datadictionary responds with the Output Panel which shows the results of the SQL processing.
- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press the Clear key to return to the Interactive SQL Service Facility Panel where you can choose another option.



**Impact**

The results of processing the DROP TABLE have a far reaching impact.

- The table data is deleted and the space is reclaimed.
- All views and synonyms based on the table are dropped.
- All application plans and statements that reference the table are invalidated.
- All columns, keys, elements, and support data associated with the table are also obsoleted (columns appear as FIELD occurrences in CA Datacom Datadictionary ). The support data includes aliases, descriptors, relationship definitions, and text (SQL comments and text added through CA Datacom Datadictionary ).
- If the table definition also exists in TEST or HIST status in CA Datacom Datadictionary, those substructure status/versions, including their elements, keys and fields, are deleted at the same time as the PRODUCTION status.

**Example Source Member**

The following example shows the DROP TABLE statement. See How to Submit SQL Statements for the steps to obtain the panel and an explanation of the fields on the panel.

```

=>
=>
=>

-----
Interactive SQL Service Facility                               SQLMAINT
                                     Source Panel                S01S
      EDIT      Member: DTABLE   Output Line Limit: 01000
                Person: JONES
                Current Authid: JONES
                Description: DROP TABLE DEPTTBL
-----
===== T O P =====
000001 drop table jones.depttbl;
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD  PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT    PF12=ALTERNATE

```

The DROP statement is required, as is the keyword TABLE. The keyword must be followed by the name of the object you are dropping.

Once this source member is executed, you cannot recall the table definition. If you want to use this table again, you must re-create it.

See [DROP](#) (see page 725) for more information on the syntax of the DROP statement.

After placing your SQL statement in the numbered line area on the Source Panel, you can perform the following:

- Enter additional SQL statements. Place a semicolon (;) after each complete statement.
- Press PF9 (EXECUTE) to execute the source member. CA Datacom Datadictionary responds with the Output Panel which shows the results of the SQL processing.
- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press the Clear key to return to the Interactive SQL Service Facility Panel where you can choose another option.

#### Example Output Member

When you execute the statement, you receive a Output Panel similar to the following example:

```
=>
=>
=>

----- >>>
Interactive SQL Service Facility                               SQLMAINT

                    Output Panel                              S010
             EDIT      Member= DTABLE
             Description: DROP TABLE DEPTTBL
-----
===== T O P =====
000001 (OUTPUT CREATED FROM SOURCE MEMBER DTABLE )
000002 drop table jones.depttbl;
000003 -----+-----+-----+-----+-----+-----+-----+-----+
000004 STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
000005 -----+-----+-----+-----+-----+-----+-----+-----+
000006 SQL STATEMENTS ASSUMED TO BE BETWEEN COLUMNS 1 AND 72
000007 NUMBER OF INPUT RECORDS READ IS 0001
000008 NUMBER OF SQL STATEMENTS PROCESSED IS 0001
===== B O T T O M =====
PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD   PF8=FORWARD
PF9=EXECUTE   PF10=LEFT     PF11=RIGHT     PF12=ALTERNATE
```

To exit from the Output Panel, you can perform one of the following:

- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press PF12 (ALTERNATE) to display the Source Panel.
- Press the Clear key to return to the Interactive SQL Service Facility Panel.

## Dropping a View

Use the DROP VIEW statement to obsolete a view in CA Datacom Datadictionary. When you drop a view, the synonyms or views that are directly or indirectly dependent on the view are automatically dropped. All application plans and statements that reference the view are invalidated.

Enter the DROP VIEW statement in the numbered area of an EDIT Source Member Panel. See [How to Submit SQL Statements](#) (see page 358) for the steps to obtain the panel and an explanation of the fields on the panel.

### **Specifying Authorization ID**

To be certain you are dropping the desired view, qualify the view name with the authorization ID in the DROP statement. If you do not specify the authorization ID, CA Datacom Datadictionary uses the current default authorization ID for the online session.

### Example Source Member

The following example shows the DROP VIEW statement. See [How to Submit SQL Statements](#) (see page 358) for the steps to obtain the panel and an explanation of the fields on the panel.

```

=>
=>
=>

-----
Interactive SQL Service Facility                                SQLMAINT
                                     Source Panel                S01S
EDIT      Member: DVIEW      Output Line Limit: 01000
          Person: JONES
          Current Authid: JONES
          Description: DROP VIEW DEPTVIEW
-----
===== T O P =====
000001 drop view jones.deptview;
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD  PF8=FORWARD
PF9=EXECUTE   PF10=LEFT     PF11=RIGHT    PF12=ALTERNATE

```

The DROP statement is required, as is the keyword VIEW. The keyword must be followed by the name of the object you are dropping.

Once this source member is executed, you cannot recall the view definition. If you want to use this view again, you must re-create it.

See [DROP](#) (see page 725) for more information on the syntax of the DROP statement.



```
=>
=>
=>

-----
Interactive SQL Service Facility                               SQLMAINT
                                     Source Panel                S01S
EDIT      Member: DSYNONYM  Output Line Limit: 01000
          Person: JONES
          Current Authid: JONES
          Description: DROP SYNONYM PROJECTS
-----
===== T O P =====
000001 drop synonym projects;
===== B O T T O M =====

PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM    PF7=BACKWARD  PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT    PF12=ALTERNATE
```

The DROP statement is required, as is the keyword SYNONYM. The keyword must be followed by the name of the object you are dropping.

Once this source member is executed, you cannot recall the synonym definition. If you want to use this synonym again, you must re-create it.

See [DROP](#) (see page 725) for more information on the syntax of the DROP statement.

After placing your SQL statement in the numbered line area on the Source Panel, you can perform the following:

- Enter additional SQL statements. Place a semicolon (;) after each complete statement.
- Press PF9 (EXECUTE) to execute the source member. CA Datacom Datadictionary responds with the Output Panel which shows the results of the SQL processing.
- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press the Clear key to return to the Interactive SQL Service Facility Panel where you can choose another option.



**Example Output Member**

When you execute the statement, you receive a Output Panel similar to the following example:

```

=>
=>
=>

----- >>>
Interactive SQL Service Facility                               SQLMAINT

                                Output Panel                S010
          EDIT     Member= DSYNONYM
                Description: DROP SYNONYM PROJECTS
-----
===== T O P =====
000001 (OUTPUT CREATED FROM SOURCE MEMBER DSYNONYM)
000002 drop synonym projects;
000003 -----+-----+-----+-----+-----+-----+-----+---
000004 STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
000005 -----+-----+-----+-----+-----+-----+-----+---
000006 SQL STATEMENTS ASSUMED TO BE BETWEEN COLUMNS 1 AND 72
000007 NUMBER OF INPUT RECORDS READ IS 0001
000008 NUMBER OF SQL STATEMENTS PROCESSED IS 0001
===== B O T T O M =====
PF1=HELP       PF2=END       PF3=SPLIT      PF4=PROCESS
PF5=TOP        PF6=BOTTOM    PF7=BACKWARD  PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT    PF12=ALTERNATE

```

To exit from the Output Panel, you can perform one of the following:

- Press PF2 (END) to obtain an SQLMAINT Panel where you can select another option.
- Press PF12 (ALTERNATE) to display the Source Panel.
- Press the Clear key to return to the Interactive SQL Service Facility Panel.



# Chapter 19: Manipulating Data in SQL Tables

---

In the Interactive SQL Service Facility, you can use the following SQL Data Manipulation Language (DML) statements to insert, update, or delete data in SQL tables in the production database. See [How to Submit SQL Statements](#).

- **Searched DELETE**

Deletes rows that meet a specified condition from a table or view. Deleting a row from a view deletes the row from the table on which the view is based. See [DELETE](#) (see page 717) for the statement syntax and more information.
- **INSERT**

Inserts a row into a table or view. Inserting a row into a view inserts the row into the table on which the view is based. See [INSERT](#) (see page 747) for the statement syntax and more information.
- **SELECT**

Produces a result table consisting of qualifying rows from the specified table. See [SELECT](#) (see page 766) for the statement syntax and more information.
- **Searched UPDATE**

Updates rows that meet a specified condition from a table or view. Updating a row in a view updates the row in the table on which the view is based. See [UPDATE](#) (see page 789) for the statement syntax and more information.

You can also perform the transaction level operations with the following Data Manipulation Language commands.

- **COMMIT WORK**

Terminates a unit of recovery and commits the database changes that were made by that unit of recovery. See [COMMIT WORK](#) (see page 616) for the statement syntax and more information.
- **LOCK TABLE**

Ensures against repeatable reads and provides an isolation level protection from other executing programs. See [LOCK TABLE](#) (see page 751) for the statement syntax and more information.

- ROLLBACK WORK

Terminates a unit of recovery and backs out the CA Datacom/DB database changes made by that unit of recovery. See [ROLLBACK WORK](#) (see page 764) for the statement syntax and more information.

**Note:** The COMMIT WORK and ROLLBACK WORK commands can be placed in a source member to be executed, if necessary. A COMMIT WORK command is implied, in CICS, by a transaction boundary (that is to say, from EXECUTE to display of results is a single transaction). A ROLLBACK WORK is implied, in all environments, by a negative SQL return code (in the format -nnn) on any SQL statement in the member being executed.

# Chapter 20: Controlling Access Through SQL Statements

---

In the Interactive SQL Service Facility, you can use the SQL Data Control Language (DCL) to define the kind of access authorized for a particular resource. This includes granting privileges to other users or revoking those privileges. See [How to Submit SQL Statements](#) (see page 358).

- GRANT

Gives specified privileges on tables and views in databases that are secured under the SQL Security Model. See [GRANT](#) (see page 742) for the statement syntax and more information.

- REVOKE

Removes privileges on tables and views for which privileges have been given through the GRANT statement. See [REVOKE](#) (see page 759) for the statement syntax and more information.

**Important!** If the CA Datacom/DB Security Facility is not installed at your site, the GRANT and REVOKE statements are rejected with an SQL error code -559. All tables and views in the statement must belong to databases secured under the SQL Security Model. If tables and views are in databases secured under the CA Datacom/DB External Security Model, the GRANT and REVOKE statements are rejected with an SQL error code -273. See the *CA Datacom Security Reference Guide* for more information about security models.



# Chapter 21: Performing SQL Administrative Tasks

---

You can use the SQL Administrative functions to perform the following maintenance:

- Set a default authorization ID for the current session.
- Rebind a plan.
- Delete a plan.

The SET AUTHID option allows you to change the default authorization ID used for the execution of SQL members in your current Interactive SQL Service Facility session. This default is released when you exit from the Interactive SQL Service Facility.

The plan options are for administration of plans associated with programs. You can rebind plans without having to re-compile, and delete plans when the associated program becomes obsolete.

## SQL Names

The authorization ID and plan names used in this facility are the SQL names, not the CA Datacom Datadictionary occurrence names. The names can be up to 18 characters in length. See [Naming the Plan](#) (see page 209) for information on how a plan name is created as a result of an application program.

## Setting the Session Authorization ID

An SQL authorization ID is a entity-occurrence of the AUTHORIZATION entity-type which was created using a CREATE SCHEMA statement. An AUTHORIZATION occurrence created through another mode of CA Datacom Datadictionary or through batch transactions cannot be an SQL authorization ID.

When the AUTHORIZATION occurrence is created (using CREATE SCHEMA), the AUTH-USAGE attribute is assigned a value of S, indicating this occurrence is an SQL authorization ID.

An AUTHORIZATION occurrence marked for use by SQL can be related to one or more PERSON occurrences (user signon ID) with the PER-ATZ-AUTHID relationship. This is a *many-to-one* relationship, meaning that several PERSON occurrences can be related to a specific AUTHORIZATION occurrence. Several users can have the same default, but each user can have only one default. See [Relating the Person to the AUTHID](#) (see page 401).

**Important!** Do not relate a PERSON occurrence to the SYSUSR AUTHORIZATION occurrence using the PER-ATZ-AUTHID relationship.

## Current Authorization ID at Start of Session

When you enter the Interactive SQL Service Facility of online CA Datacom Datadictionary, your default authorization ID is one of the following:

- An AUTHID established with CA Datacom Datadictionary functions to relate your PERSON occurrence to an SQL valid AUTHORIZATION occurrence using the PER-ATZ-AUTHID relationship. (For information on adding relationships between occurrences, see the *CA Datacom Datadictionary Batch Reference Guide* or *CA Datacom Datadictionary Online Reference Guide*.)
- An AUTHID established with the Default Authorization Panel or the RESET DEFAULT AUTHID command. (See [Changing Your AUTHID](#) (see page 402).)

The SET AUTHID command in the SQLADMIN facility allows you to temporarily change the default authorization ID during an online session in the Interactive SQL Service Facility. Changing the default is useful when you want to process DML statements against tables belonging to another schema without having to qualify those tables with the authorization ID.



**Note:** This option **does not** create a new authorization ID. If you want to create a new authorization ID, see [Creating a Schema](#) (see page 400).

The default authorization ID is the one CA Datacom Datadictionary uses with any executions of SQL members in the Interactive SQL Service Facility. If the SQL statements in the source member being executed do not make an explicit reference to an authorization ID, CA Datacom Datadictionary uses the current default authorization ID.

You can switch to any valid SQL authorization ID. If the specified authorization ID is not valid, the switch does not take place and the current authorization ID in effect remains intact for the session.

## Displaying and Reporting

You can display a table of AUTHORIZATION entity-occurrences using the Entity Display (ENTDISPL) Mode of CA Datacom Datadictionary online and view the attributes to determine if AUTH-USAGE=S. This indicates it is a valid SQL authorization ID. See the *CA Datacom Datadictionary Online Reference Guide* for information on the Entity Display Mode.

You can also run a batch report using the -RPT SCHEMA transaction to list all valid SQL authorization IDs and objects owned by the schema represented by each AUTHID. See the *CA Datacom Datadictionary Batch Reference Guide* for information on the Schema Report.

## How to Set the Default

Use the following steps to set the default authorization ID for a session.

### Step 1

When you sign on or select the SET MODE function, CA Datacom Datadictionary displays the Datadictionary Mode Select Panel. Select Option 7 or enter the SET MODE SQL command.

### Step 2

On the Interactive SQL Service Facility Panel, select Option 2 (SQLADMIN).

```
=>
=>
=>
-----
Interactive SQL Service Facility

Enter desired option number ==> __ (There are 05 options on the menu)
1. SQLMAINT          SQL member maintenance/execution
2. SQLADMIN          SQL administrative functions
3. SET MODE          Reset Datadictionary processing mode
4. IDEAL             Transfer to IDEAL application
5. OFF              End session
```

**Note:** If CA Ideal is not installed on your system, the OFF option appears in place of the CA Ideal option on the panel.

### Step 3

CA Datacom Datadictionary displays the Interactive SQL Service Facility SQLADMIN Panel with four menu options. Select Option 1 (SET AUTHID).

```
=>
=>
=>
-----
Interactive SQL Service Facility                               SQLADMIN

Enter desired option number ==> __ (There are 05 options on the menu)
1. SET AUTHID        Set default AUTHID for session
2. DELETE PLAN      Delete SQL plan
3. REBIND PLAN      Rebind SQL plan
4. DISPLAY PLAN     Display index of SQL plans
5. END              End SQLADMIN processing
```

**Step 4**

The AUTHORIZATION-ID Panel appears. After reading the following information, complete your entry on the panel and press PF9 (EXECUTE). To change to the default authorization ID, press PF9 (EXECUTE) without placing an entry on the panel.

```

=>
=>
=>
-----
Interactive SQL Service Facility                                SQLADMIN
                                                                S03U

                                AUTHORIZATION-ID

ENTER AUTHORIZATION-ID:  _____

PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD  PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT    PF12=ALTERNATE

```

**Note:** If your terminal displays 24 lines, you must scroll forward to display the last line containing the descriptions of PF9 (EXECUTE) through PF12 (ALTERNATE).

**ENTER AUTHORIZATION-ID**

*(Optional)* Enter the authorization ID which you want as the default for this session of the Interactive SQL Service Facility.

If you leave this field blank, CA Datacom Datadictionary resets the current authorization ID to the default which was in effect when you signed on or that you established with the RESET DEFAULT AUTHID command (see [Changing Your AUTHID](#) (see page 402)).

**Valid Entries:**

1- to 18-character authorization ID

**Default Value:**

Authorization ID in effect at signon

**Step 5**

When you successfully execute the request, CA Datacom Datadictionary returns a message to the panel naming the default authorization ID in effect. In this example, the default authorization is set to JONES.

```
⇒
⇒
⇒
-----
Interactive SQL Service Facility                                SQLADMIN
                                                                S03U
                                AUTHORIZATION-ID

ENTER AUTHORIZATION-ID:   JONES_____
AUTHORIZATION-ID IS NOW JONES

PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD  PF8=FORWARD
PF9=EXECUTE   PF10=LEFT   PF11=RIGHT    PF12=ALTERNATE
```

To exit this panel, you can:

- Press PF2 (END) to display the Interactive SQL Service Facility SQLADMIN Panel.
- Press the CLEAR key to return to the Interactive SQL Service Facility Panel where you can choose another option.

## Deleting a Plan

The DELETE PLAN option of the SQLADMIN facility allows you to remove plans that, for example, are associated with an application that is no longer in use or is invalid and you want to reuse the plan name. See [Preliminary Information—Lock Levels](#) (see page 597) for information about lock levels with regard to DELETE PLAN.

## How the Plan Is Named

See [Naming the Plan](#) (see page 209) for information on how a plan name is created as a result of an application program.

When a source member is created and executed, CA Datacom Datadictionary creates and deletes a plan for that member during the execution. If the execution of the member is successful, the plan is deleted. If you execute the source member again, CA Datacom Datadictionary creates and deletes a new plan for that member.

The name for the plan is formed by concatenating the character string SQLD with the four-byte terminal ID. For example, if the terminal ID is WXYZ, the plan name for a source member executed by the user is SQLDWXYZ.

In certain circumstances, the plan for a source member may not be automatically deleted. For example, if the CA Datacom/DB MUF terminates abnormally while a source member is executing, there is a possibility of the plan not being deleted.

An automatic plan deletion can also fail when CA Datacom Datadictionary issues an internal command to delete the plan. In this case, a message specifying that the command failed is displayed on the output panel after the line stating NUMBER OF INPUT RECORDS READ IS.

If you attempt to execute a source member and receive the message PLAN ALREADY EXISTS, the automatic plan delete failed after a previous execution. You must delete the plan before you can execute the source member.

You can display an index report of PLAN occurrences using the Entity Display (ENTDISPL) Mode of CA Datacom Datadictionary or the display plan command in the SQLADMIN mode. See the *CA Datacom Datadictionary Online Reference Guide* or *CA Datacom Datadictionary User Guide*. Also, you can produce an index report for the PLAN entity-type using CA Datacom Datadictionary batch reporting. See the *CA Datacom Datadictionary Batch Reference Guide*. If the name of any PLAN occurrences in the index report begins with SQLD, the automatic deletion of online plans did not take place and the plan must be manually deleted.

## How to Delete a Plan

Use the following steps to delete a plan.

### Step 1

When you sign on or select the SET MODE function, CA Datacom Datadictionary displays the Datadictionary Mode Select Panel. Select Option 7 or enter the SET MODE SQL command.

### Step 2

On the Interactive SQL Service Facility Panel, select Option 2 (SQLADMIN).

### Step 3

CA Datacom Datadictionary displays the Interactive SQL Service Facility SQLADMIN Panel with five menu options. Select Option 2 (DELETE PLAN).

```
=>
=>
=>
-----
Interactive SQL Service Facility                               SQLADMIN

Enter desired option number ==> _ (There are 05 options on the menu)
 1. SET AUTHID                               Set default AUTHID for session
 2. DELETE PLAN                              Delete SQL plan
 3. REBIND PLAN                              Rebind SQL plan
 4. DISPLAY PLAN                             Display index of SQL plans
 5. END                                       End SQLADMIN processing
```

**Step 4**

The DELETE PLAN Panel appears. After reading the list below, complete your entries on the panel and press PF9 (EXECUTE) to execute your request.

```

=>
=>
=>
-----
Interactive SQL Service Facility                                SQLADMIN
                                DELETE PLAN PANEL                S03U

ENTER PLAN NAME TO DELETE : _____
ENTER AUTHORIZATION-ID:    _____

PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD   PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT     PF12=ALTERNATE

```

**ENTER PLAN NAME TO DELETE**

*(Required)* Specify the name of the plan you want to delete. This is the name specified in the SQL PLANAME= option or the PROGRAM-ID value in the application program.

**Valid Entries:**

1- to 18-character plan name

**Default Value:**

(No default)

**ENTER AUTHORIZATION-ID**

*(Optional)* Specify authorization ID which owns the plan you are deleting.

If you do not specify the authorization ID, CA Datacom Datadictionary uses the default authorization ID for the current online session.

It is possible for different authorization IDs to have a plan with the same name. To be certain you are deleting the right plan, specify authorization ID on this panel if you do not know the default for current online session.

**Valid Entries:**

1- to 18-character authorization ID

**Default Value:**

Default authorization ID for current session

**Step 5**

When you successfully execute the request, CA Datacom Datadictionary returns a message to the panel naming the plan and the authorization-ID, which is either the one you specified on the panel or the default for the session if you left that field blank.

```
=>
=>
=>

-----
Interactive SQL Service Facility                                SQLADMIN
                                DELETE PLAN PANEL                                S03U

ENTER PLAN NAME TO DELETE : PLANA_____
ENTER AUTHORIZATION-ID: _____ AUTHID=JONES
SUCCESSFUL DELETE OF PLAN PLANA_____ AUTHID=JONES

PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD  PF8=FORWARD
PF9=EXECUTE   PF10=LEFT   PF11=RIGHT    PF12=ALTERNATE
```

After successfully deleting the plan, you can:

- Press PF2 (END) to display the Interactive SQL Service Facility SQLADMIN Panel.
- Press the CLEAR key to return to the Interactive SQL Service Facility Panel where you can choose another option.



## Rebinding a Plan

The Interactive SQL Service Facility allows you to specify plan options when rebinding an existing plan that was created as a result of SQL statements embedded in a program. The plan options you can specify during the Interactive SQL Service Facility session are slightly different from those you can specify before submitting a program with embedded SQL statements. See [Coding Plan Options](#) (see page 464).

A change to an SQL table definition marks any plans referencing that table for a rebind. CA Datacom/DB attempts an automatic rebind of the plan when the program executes. If the rebind fails, you must make any necessary changes to your program and submit it to the SQL Preprocessor.

The REBIND PLAN option allows you to manually control the rebind instead of waiting for the automatic rebind attempt. A manual rebind:

- Saves execution time resources.
- Identifies source program changes necessary before the next scheduled execution of that program.
- Allows you to change the plan options.
- Could prevent abnormal termination of the program.

The REBIND PLAN option allows you to update a plan that may have become invalid but is assumed to be successfully rebound without the need for precompiling and linking the program again.

## How to Rebind a Plan

Use the following steps to rebind a plan:

### Step 1

When you sign on or select the SET MODE function, CA Datacom Datadictionary displays the Datadictionary Mode Select Panel. Select Option 7 or enter the SET MODE SQL command.

### Step 2

On the Interactive SQL Service Facility Panel, select Option 2 (SQLADMIN).

### Step 3

CA Datacom Datadictionary displays the Interactive SQL Service Facility SQLADMIN Panel. Select Option 3 (REBIND PLAN).

```

=>
=>
=>
-----
Interactive SQL Service Facility                                SQLADMIN

Enter desired option number ==> _ (There are 05 options on the menu)
 1. SET AUTHID                Set default AUTHID for session
 2. DELETE PLAN               Delete SQL plan
 3. REBIND PLAN               Rebind SQL plan
 4. DISPLAY PLAN              Display index of SQL plans
 5. END                       End SQLADMIN processing
    
```

### Step 4

The REBIND PLAN Panel appears. After reading the list below, complete your entries on the panel and press PF9 (EXECUTE) to execute your request.

The options for the plan are displayed on the panel.

```

=>
=>
=>
-----
Interactive SQL Service Facility                                SQLADMIN

                                REBIND PLAN PANEL                                S03U

ENTER PLAN NAME TO REBIND : _____
ENTER AUTHORIZATION-ID:      _____

                                PLAN OPTIONS
CBSIO   : _____ (between 0 and 524287)   MSG-PREP: _ (N,S or D)
PRIORITY: _ (between 1 and 15)               MSG-EXEC: _ (N,S or D)
TIMEMIN : ___ (between 0 and 120)            PLNCLOSE: _ (T or R)
TIMESEC  : ___ (between 0 and 120)            PLNISOLA: _ (U,C or R)
SQLMODE  : _____ (DATACOM, ANSI, FIPS)   PLNJOVRD: _ (M,P or E)
PLNWKSP  : _____ (between 0 and 0128K)   PLNDATE  : _ (0 thru 4)
PLNTIME  : _ (0 thru 4)

PF1=HELP   PF2=END     PF3=SPLIT   PF4=PROCESS
PF5=TOP    PF6=BOTTOM  PF7=BACKWARD PF8=FORWARD
PF9=EXECUTE PF10=LEFT   PF11=RIGHT  PF12=ALTERNATE
    
```

**ENTER PLAN NAME TO REBIND**

*(Required)* Specify the name of the plan you want to rebind. This is the name specified in the SQL PLANAME= option or the PROGRAM-ID value in the application program. See [Naming the Plan](#) (see page 209).

**Valid Entries:**

1- to 18-character plan name

**Default Value:**

(No default)

**ENTER AUTHORIZATION-ID**

*(Optional)* Specify the authorization ID which owns the plan you are rebinding. If you do not specify the authorization ID, CA Datacom Datadictionary uses the default authorization ID for the current online session.

It is possible for different authorization IDs to have a plan with the same name. To make certain you are rebinding the right plan, specify the authorization ID on this panel if you do not know the default for the current online session.

**Valid Entries:**

1- to 18-character authorization ID

**Default Value:**

Default authorization ID for current session

**PLAN OPTIONS**

The options you can specify in the rebind of this plan are shown on the panel. See [Coding Plan Options](#) (see page 464) for an explanation of the valid entries.

**Step 5**

When you successfully execute the request, CA Datacom Datadictionary returns a message to the panel naming the plan and the authorization-ID. The AUTHID is either the one you specified on the panel or the default for the session if you left that field blank.

```

=>
=>
=>
-----
Interactive SQL Service Facility                                SQLADMIN
                                REBIND  PLAN PANEL                                S03U

ENTER PLAN NAME TO REBIND : PLANB
ENTER AUTHORIZATION-ID:      JONES
SUCCESSFUL REBIND OF PLAN PLANB                                AUTHID=JONES

                                PLAN OPTIONS
CBSIO   : _____ (between 0 and 524287)    MSG-PREP: _ (N,S or D)
PRIORITY: _ (between 1 and 15)                MSG-EXEC: _ (N,S or D)
TIMEMIN : _ (between 0 and 120)               PLNCLOSE: _ (T or R)
TIMESEC : _ (between 0 and 120)               PLNISOLA: _ (U,C or R)
SQLMODE : _____ (DATACOM, ANSI, FIPS)    PLNJOVRD: _ (M,P or E)
PLNWKSP : _ (between 0 and 0128K)            PLNDATE : _ (0 thru 4)
PLNTIME : _ (0 thru 4)

PF1=HELP      PF2=END      PF3=SPLIT      PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD   PF8=FORWARD
PF9=EXECUTE   PF10=LEFT    PF11=RIGHT     PF12=ALTERNATE
  
```

After successfully rebinding the plan, you can:

- Press PF2 (END) to display the Interactive SQL Service Facility SQLADMIN Panel.
- Press the CLEAR key to return to the Interactive SQL Service Facility Panel where you can choose another option.

**Note:** If you press Enter instead of PF9 (EXECUTE), the values for each option the plan currently has are displayed. If you press Enter instead of PF9 (EXECUTE) after making a change to any option, the current option is redisplayed.

## Displaying Index of SQL Plans

The Interactive SQL Service Facility allows you to display an index of all plans that currently exist in the dictionary.

You receive an error message telling you that no plans currently exist if there are none, or if you are not pointing to the active dictionary where the plan occurrences are defined.

**Note:** The online plan is usually deleted after successful execution. The DISPLAY PLAN option should therefore only show the plans created by embedded SQL in application programs. However, if an online plan is not deleted for some reason, such as an abend, it is displayed.

## How to Display an Index of SQL Plans

Use the following steps to display an index of SQL plans:

### Step 1

When you sign on or select the SET MODE function, CA Datacom Datadictionary displays the Datadictionary Mode Select Panel. Select Option 7 or enter the SET MODE SQL command.

### Step 2

On the Interactive SQL Service Facility Panel, select Option 2 (SQLADMIN).

### Step 3

CA Datacom Datadictionary displays the Interactive SQL Service Facility SQLADMIN Panel. Select Option 4 (DISPLAY PLAN).

```
=>
=>
=>
-----
Interactive SQL Service Facility                                SQLADMIN

Enter desired option number ==> _ (There are 05 options on the menu)
 1. SET AUTHID                               Set default AUTHID for session
 2. DELETE PLAN                              Delete SQL plan
 3. REBIND PLAN                               Rebind SQL plan
 4. DISPLAY PLAN                             Display index of SQL plans
 5. END                                       End SQLADMIN processing
```

### Step 4

The following display appears.

```

=>
=>
=>

----- >>>
DDOL: ANCHORED OCCURRENCE FOR INDEX DISPLAY

TYPE OCCURRENCE STA VER S04D
PLN SYSADM-BANEMODE P 001

-----
Type Occurrence-Name Status Version
===== T O P =====
000001 PLN SYSADM-BANEMODE P 001
000002 PLN SYSADM-CANEMODE P 001
000003 PLN SYSADM-DANEMODE P 001
000004 PLN SYSADM-EANEMODE P 001
000005 PLN SYSADM-LANEMODE P 001
000006 PLN SYSADM-TANEMODE P 001
000007 PLN SYSADM-TESTMODE P 001
===== B O T T O M =====
PF1=HELP PF2=END PF3=SPLIT PF4=PROCESS
PF5=TOP PF6=BOTTOM PF7=BACKWARD PF8=FORWARD
PF9=EXECUTE PF10=LEFT PF11=RIGHT PF12=ALTERNATE
    
```

After a successful display of the index of SQL plans, you can:

- Press PF2 (END) to display the Interactive SQL Service Facility SQLADMIN Panel.
- Press the CLEAR key to return to the Interactive SQL Service Facility Panel where you can choose another option.

If no plans currently exist, or if you are not pointing to the active dictionary where the plan occurrences are defined, you receive an error message as shown following instead of the index display.

```

=>
=>
=>
1-DDOL000472I - STRP - NO PLANS CURRENTLY EXIST ON DICTIONARY

-----
Interactive SQL Service Facility SQLADMIN
Enter desired option number ==> (There are 05 options on the menu)
1. SET AUTHID Set default AUTHID for session
2. DELETE PLAN Delete SQL plan
3. REBIND PLAN Rebind SQL plan
4. DISPLAY PLAN Display index of SQL plans
5. END End SQLADMIN processing
    
```

## Specifying Plan Options in a Source Member

The plan is controlled by options which you can specify or allow to default. The values you assign to the options determine how the plan processes the SQL statements and controls certain aspects of the environment.

The options you specify in your source member build the CA Datacom/DB access plan containing the SQL statements you place in a source member. This access plan is temporary and is deleted after the SQL statements are executed. The plan only continues to exist if the system abends before the plan is deleted. The access plan contains:

- Information required by CA Datacom/DB about your source member.
- Each embedded SQL statement in your source member.

**Note:** The plan options you specify in your source member have no affect on SQL statements submitted through CA Dataquery.

The plan options you code in a source member must appear before your SQL statements. The following example shows the format:

```

=>
=>
=>

-----
Interactive SQL Service Facility                               SQLMAINT
                                     Source Panel                S01S
      EDIT      Member: $DDSQL   Output Line Limit:  01000
                Person: JONES
      Current Authid: JONES
      Description: CREATE TABLE DEPTTBL
-----
===== T O P =====
000001 *$dbsqlopt prty=9;
..... create table depttbl
..... (deptno char(2) not null,
..... deptname char(24) not null,
..... mgrnbr char(6) not null,
..... admdept char(2),
..... unique (deptno, mgrnbr));
PF1=HELP      PF2=END      PF3=SPLIT     PF4=PROCESS
PF5=TOP       PF6=BOTTOM   PF7=BACKWARD PF8=FORWARD
PF9=EXECUTE   PF10=LEFT   PF11=RIGHT    PF12=ALTERNATE

```

When you code the options in your source member, you can enter the options in any order, but the following rules apply:

- The options must be the first statement in your source member and must begin with \*\$DBSQLOPT.
- No option can be continued from one line to the next, that is to say, the option keyword and assigned value must be coded on the same line.
- Each option must be separated from another by one space.
- Each option you code must be in the option-name=value format.
- The last option keyed must be followed by a semicolon (;).

All options which you enter in the source member are edited. The default value for an option is used if you do not specify the option in your member.

If you enter the option keyword, but do not specify a value after the equal sign or if an invalid value is specified for an option or the option itself is misspelled, you receive error code DDOL000008 -- PLAN OPTION(S) IN ERROR.

## Coding Plan Options

The following options are valid only when submitted with a program with embedded SQL. You cannot change them on the REBIND PLAN panel. For more information on options submitted with a program with embedded SQL, see [Options You Can Specify](#) (see page 214).

### Valid Options per Language:

COBOL	PL/I	C	Assembler
APOST=			
AUTHID=	AUTHID=	AUTHID=	AUTHID=
CBSIO=	CBSIO=	CBSIO=	CBSIO=
CHECKPLAN=	CHECKPLAN=	CHECKPLAN=	CHECKPLAN=
CHECKWHEN=	CHECKWHEN=	CHECKWHEN=	CHECKWHEN=
CHECKWHO=	CHECKWHO=	CHECKWHO=	CHECKWHO=
COBMODE=			
DATE=	DATE=	DATE=	DATE=
DECPOINT=	DECPOINT= DECPT=	DECPOINT= DECPT=	
GENSECTN=			



<b>COBOL</b>	<b>PL/I</b>	<b>C</b>	<b>Assembler</b>
	GENSTOR=		GENSTOR=
	GENINIT=		GENINIT=
			INLINE=
ISOLEVEL=	ISOLEVEL=	ISOLEVEL=	ISOLEVEL=
ITYP=	ITYP=	ITYP=	ITYP=
	LANGUAGE= LANG=	LANGUAGE= LANG=	LANGUAGE= LANG=
	MARGINS=		MARGINS=
MSG=			
	MSGEXEC=	MSGEXEC=	MSGEXEC=
	MSGPREC=	MSGPREC=	MSGPREC=
OPT=	OPT=	OPT=	OPT=
PAGESIZE=	PAGESIZE=	PAGESIZE=	PAGESIZE=
PLANAME=	PLANAME= PLANNAME=	PLANAME= PLANNAME=	PLANAME= PLANNAME=
PLNCLOSE=	PLNCLOSE=	PLNCLOSE=	PLNCLOSE=
PRTREXIT=	PRTREXIT=	PRTREXIT=	PRTREXIT=
PRTY=	PRTY=	PRTY=	PRTY=
QUOTE=			REFNTRY=
SAVEPLANSEC=	SAVEPLANSEC=	SAVEPLANSEC=	SAVEPLANSEC=
	SMBR=	SMBR=	SMBR=
SQLMODE=	SQLMODE=	SQLMODE=	SQLMODE=
STRDELIM=	STRDELIM= STRDLM= STRINGDELIM=	STRDELIM= STRDLM= STRINGDELIM=	
TIME=	TIME=	TIME=	TIME=
TIMEMIN=	TIMEMIN=	TIMEMIN=	TIMEMIN=
TIMESEC=	TIMESEC=	TIMESEC=	TIMESEC=
	UCRPT=		UCRPT=
USRNTRY=			USRNTRY=

COBOL	PL/I	C	Assembler
VIEWSEC=	VIEWSEC=	VIEWSEC=	VIEWSEC=
WORKSPACE=	WORKSPACE=	WORKSPACE=	WORKSPACE=

The following options are valid only when submitted in a source member with a \*\$DBSQLOPT statement or with a program with embedded SQL. You cannot change them on the REBIND PLAN panel:

- DECPOINT= (in COBOL, PL/I, and C only)
- STRDELIM= (in COBOL, PL/I, and C only)

For more information about DECPOINT= and STRDELIM= see [Options You Can Specify](#) (see page 214).

You can specify alternate values for the following plan options in the Interactive SQL Service Facility.

Plan Option on Rebind Panel	Plan Option in Source Member	Preprocessor Option (see chapter 4) *
CBSIO	CBSIO=	CBSIO=
MSG-EXEC MSG-PREP	PLNMSGPE=	MSG= (COBOL) MSGEXEC= (PL/I, C, Assembler) MSGPREC= (PL/I, C, Assembler)
PLNCLOSE	PLNCLOSE=	PLNCLOSE=
PLNDATE	PLNDATE=	DATE=
PLNISOLA	PLNISOLA=	ISOLEVEL=
PLNJOVRD	PLNJOVRD=	OPT=
PLNTIME	PLNTIME=	TIME=
PLNWKSP	PLNWKSP=	WORKSPACE=
PRIORITY	PRTY=	PRTY=
SQLMODE	SQLMODE=	SQLMODE=
TIMEMIN	TIMEMIN=	TIMEMIN=
TIMESEC	TIMESEC=	TIMESEC=

\* In the Preprocessor column, except where noted the listed options are for COBOL, PL/I, C, and Assembler.

## Options You Can Specify

You can specify values for plan options on the REBIND PLAN panel or in a source member with a \$DBSQLOPT statement. When you do not specify a value, the value specified for the existing plan remains in effect.

### **CBSIO=**

Specifies an I/O limit interrupt value for all SQL commands that create a set. This option allows application environments to establish their own maximums in I/O and set processing relative to their own requirements.

You can use this option to limit the computer resources that can be used for each execution of the following statements in the plan:

- OPEN CURSOR, FETCH CURSOR
- SELECT INTO
- INSERT, UPDATE, DELETE
- CREATE INDEX, DROP INDEX, ALTER TABLE

For cursors, the limit applies to the total resources used to OPEN and FETCH all rows of the cursor.

A counter is incremented each time a different index or data block is accessed, and each time 100 rows are read. Execution is terminated, and SQL return code -137 is returned when this counter exceeds the limit.

The value of the counter is reported in the Statistics and Diagnostics Area (PXX) at the end of each request to the MUF when any SQL traces are in effect.

For cursor, SELECT INTO, INSERT, UPDATE and DELETE, you can use the total estimated cost reported in the SYSADM.SYSMSG table when bind time optimization messages are requested with the MSG-PREP plan option as a guide for setting the limit. For CREATE INDEX, DROP INDEX, and ALTER TABLE, estimate the limit as the number of bytes in the table divided by 2000. You must set the limit for the most expensive statement in the plan.

A value of zero means no limit.

**Note:** Beginning in r10, here is how the CBSIO plan option is calculated: to 500,000 is added the amount over 500,000 multiplied by 10,000. For example, given a value of 500,100, the calculation would be  $500,000 + (100 * 10,000) = 500,000 + 1,000,000 = 1,500,000$ .

#### **Valid Entries:**

0—524287

#### **Default Value:**

*(in CA Datacom Datadictionary only)* 1000

### **MSG-PREP and MSG-EXEC**

Use MSG-PREP and MSG-EXEC fields on the REBIND PLAN Panel to specify the same options as the PLNMSGPE= option. MSG-PREP specifies the messages returned during preparation of the plan and MSG-EXEC specifies the messages returned during execution of the plan. Specify N for no messages, S for summary messages, or D for detail or full messages.

#### **Valid Entries:**

D, N, S

#### **Default Value:**

*(in CA Datacom Datadictionary only)* 0

### **PLNCLOSE=**

Specifies when the plan and User Requirements Table are closed.

If you specify T, the plan and User Requirements Table close when the transaction ends, that is to say, an SQL COMMIT WORK or ROLLBACK WORK statement, a CA Datacom/DB LOGCP, LOGCR or LOGTB command, or a CA Datacom CICS Services DEQUE.

We recommend the T option for a CICS environment. See OPEN/CLOSE Efficiency for CICS-related information with regard to the SQL Preprocessor's PLNCLOSE= option We also recommend PLNCLOSE=T for procedures. The T option gives the most flexibility because:

- User Requirements Tables that are opened by a plan are closed when the transaction ends, and
- Operations that require User Requirements Tables to be closed (such as a LOAD) can be performed whenever the plan is not being executed.

The R option is the most efficient, however, because:

- The User Requirements Tables, plan, and its cached statements are not closed each time the plan is not being executed by current transactions, but
- To perform operations requiring the User Requirements Tables to be closed, the CICS User Requirements Table that provides SQL access (default is URT 020) must be closed to close the plan and its User Requirements Tables.

If you specify R, the plan and User Requirements Table close when the run unit ends or when a CA Datacom/DB CLOSE command is issued. A PLNCLOSE=R plan may be rebound or precompiled without closing it, as long as it is not currently being executed by any transactions. The R option is recommended for batch programs.

#### **Valid Entries:**

T or R

#### **Default Value:**

*(in CA Datacom Datadictionary only)* T

**PLNDATE=**

Specifies the format of the date when the plan was precompiled. See the following chart for the formats:

Entry	Format	Description
0		retain existing format specification
1	yyyy-mm-dd	ISO - International Standards Organization
2	mm/dd/yyyy	USA - IBM USA Standard
3	dd.mm.yyyy	EUR - IBM European Standard
4	yyyy-mm-dd	JIS - Japanese Industrial Standard

**Valid Entries:**

0, 1, 2, 3, 4

**Default Value:**

*(in CA Datacom Datadictionary only)* 0

**PLNISOLA=**

Specifies the isolation level, or the degree to which a unit of recovery in your application is isolated from the updating operations of other units of recovery.

When you specify U, no locks are acquired for rows accessed for read-only purposes. Your application can access rows that have been updated by another unit of recovery, but the changes have not been committed.

When the value is C, for cursor stability, a unit of recovery holds locks only on its uncommitted changes and the current row of each of its cursors. This isolation level provides a high degree of concurrency.

**Note:** If you are doing updates, deletes or inserts on tables, the value must be C. Also note that if you specify ANSI or FIPS for the SQLMODE= option, ISOLEVEL= must have the value of C.

To acquire exclusive control of a table, see LOCK TABLE.

**Valid Entries:**

U or C

**Default Value:**

*(in CA Datacom Datadictionary only)* C

**PLNJOVRD=**

Specifies the join optimization sequence. Specify M if the normal join optimization is unacceptable and you want tables joined as they are listed in the FROM clause. This results in a nested loop join. Specify P to change to normal join optimization.

**Valid Entries:**

M or P

**Default Value:**

*(in CA Datacom Datadictionary only)* P

**PLNMSGPE=**

Specifies the messages returned during preparation and execution of the plan. Use the MSG-PREP and MSG-EXEC fields on the REBIND PLAN Panel to assign this value.

Specify N for no messages, S for summary messages, or D for detail or full messages. Enter a combination of letters, that is to say, SD or NS. If an S or a D is specified in the PLNMSGPE= option, the message is automatically displayed in the output member by the Interactive SQL Service Facility.

**Valid Entries:**

DD, DN, DS, ND, NN, NS, SD, SN, SS

**Default Value:**

*(in CA Datacom Datadictionary only)* NN

**PLNTIME=**

Specifies the format of the time when the plan was precompiled. See the following chart for the formats.

Entry	Format	Description
0		retain existing format specification
1	hh.mm.ss	ISO - International Standards organization
2	hh:mm AM or PM	USA - IBM USA Standard
3	hh.mm.ss	EUR - IBM European Standard
4	hh:mm:ss	JIS - Japanese Industrial Standard

**Valid Entries:**

0, 1, 2, 3, 4

**Default Value:**

*(in CA Datacom Datadictionary only)* 0

**PLNWKSP=**

**Use PLNWKSP= only at the direction of CA Support.** Specifies an increase in the amount of work space used at plan execution time.

**Valid Entries:**

0—128

**Default Value:**

*(in CA Datacom Datadictionary only)* 0

**PRTY=**

Specifies the priority of the SQL requests from the plan within the MUF. Use the PRIORITY field on the REBIND PLAN Panel to specify this option.

The lowest priority is 1, while 15 is the highest priority. If you need more information about specifying a priority, see your Database Administrator.

**Valid Entries:**

1—15

**Default Value:**

*(in CA Datacom Datadictionary only)* 7

**SQLMODE=**

Specifies the mode in which to process the program.

If you specify ANSI or FIPS, your program is processed in ANSI or FIPS mode, which means all your SQL statements must be coded according to ANSI or FIPS standards.

Names for tables, columns, views, synonyms and cursors must be 1 to 18 characters in length if SQLMODE=ANSI or SQLMODE=FIPS.

If ANSI or FIPS is specified for the mode, the PLNISOLA=U option is not allowed. PLNISOLA= must be C when SQLMODE=ANSI or SQLMODE=FIPS.

If you specify DATACOM, your program is processed in extended mode, which means CA Datacom/DB extensions to the standards are allowed in your SQL statements.

Names for tables, columns, views, synonyms and cursors can be 1 to 32 characters in length if SQLMODE=DATACOM.

Authorization IDs and plan names must be 1 to 18 characters in all modes.

**Note:** The SQLMODE MUF startup option must be set to DATACOM before this plan option is effective. If the SQLMODE MUF startup option is set to ANSI, this plan option is overridden and all SQL statements must comply with ANSI standards. If the SQLMODE MUF startup option is set to FIPS, this plan option is overridden and all SQL statements must comply with FIPS standards. See the Database Administrator for information on the value assigned to the SQLMODE Multi-User startup option.

**Valid Entries:**

ANSI, DATACOM, FIPS

**Default Value:**

(in CA Datacom Datadictionary only) DATACOM

**TIMEMIN=**

Specifies exclusive control wait time limit in minutes.

This option allows a program to either wait or not wait for an explicit amount of time when another job is holding a requested record under exclusive control. If the specified time is exceeded, the application program receives a -117 value in the SQLCODE of the SQL Communication Area and a CA Datacom/DB 61 return code to inform the user that the record was not available.

Specifying a zero for both TIMEMIN= and TIMESEC= means that there is no time limit, and without a limit on the wait time, a *wait forever* condition is possible.

TIMEMIN=0 and TIMESEC=1 means do not wait at all.

**Note:** Do not specify nonzero values for both TIMEMIN= and TIMESEC=. You can specify one or the other, not both.

If you are using CA Datacom STAR for distributed processing, see CA Datacom STAR documentation before specifying this option.

**Valid Entries:**

0—120

**Default Value:**

(in CA Datacom Datadictionary only) 0



**TIMESEC=**

Specifies exclusive control wait time limit in seconds.

This option allows a program to either wait or not wait for an explicit amount of time when another job is holding a requested record under exclusive control. If the specified time is exceeded, the application program receives a -117 value in the SQLCODE of the SQL Communication Area and a CA Datacom/DB 61 return code to inform the user that the record was not available.

Specifying a zero for both TIMEMIN= and TIMESEC= means that there is no time limit, and without a limit on the wait time, a *wait forever* condition is possible.

TIMESEC=1 and TIMEMIN=0 means do not wait at all.

**Note:** Do not specify nonzero values for both TIMEMIN= and TIMESEC=. You can specify one or the other, not both.

If you are using CA Datacom STAR for distributed processing, see CA Datacom STAR documentation before specifying this option.

**Valid Entries:**

0—120

**Default Value:**

(in CA Datacom Datadictionary only) 120

## Example

The following example of the entry area of a Source Panel shows how the plan options can be coded:

```

-----
===== T O P =====
000001 *$dbsqlopt cbsio=25000 prty=9 sqlmode=ansi
..... timesec=60;
.....      .      .
.....      .      .
.....      .      .

```



# Chapter 22: Overview of SQL Language Reference

---

The following table lists the chapters that follow and what each chapter contains:

Chapter	Contains
<a href="#">Basic Language Elements</a> (see page 477)	Describes the basic language elements of SQL, including characters, tokens, identifiers, naming conventions, authorization ID, USER, values, data types, basic operations (assignment and comparison), literals, column names, host variables and indicator variables. Includes examples for most language elements.
<a href="#">Functions</a> (see page 549)	Describes column and scalar functions and provides a syntax diagram of this basic language element, plus examples.
<a href="#">Expressions</a> (see page 527)	Describes expressions and provides a syntax diagram of this basic language element, plus examples. Also discusses expressions without arithmetic operators, with arithmetic operators and conversions between data types during arithmetic operations, and the precedence of operations.
<a href="#">Predicates</a> (see page 581)	Describes predicates, including the basic predicate, quantified predicate, BETWEEN predicate, LIKE predicate, EXISTS predicate, IN predicate and NULL predicate, and provides a syntax diagram of each predicate, plus examples.
<a href="#">Search Conditions</a> (see page 593)	Describes search conditions and provides a syntax diagram of this basic language element, plus examples. Includes information on Boolean operators.
<a href="#">SQL Statements</a> (see page 597)	Describes the various SQL statements and gives syntax diagrams and examples of each.



# Chapter 23: Basic Language Elements

---

Language elements common to many SQL statements are discussed in the following sections. Other language elements are discussed in the following chapters:

- Functions
- Expressions
- Predicates
- Search Conditions

## Characters

The basic symbols of SQL are EBCDIC characters. These include:

- **Letters:** A—Z, a, #, @, \$
- **Digits:** 0—9
- **Special Characters:** Any character other than a letter or digit

## Tokens

Tokens are the basic syntactical units of the language. A token consists of one or more characters. A token can be an ordinary token or a delimiter token. The following table lists the different types of ordinary and delimiter tokens:

Ordinary Tokens	Delimiter Tokens
Numeric literal	String literal
Ordinary SQL identifier	Operator
Host identifier	Any special character shown in syntax diagrams
Keyword	Delimited SQL identifier

## Spaces

Spaces are a sequence of one or more blank characters. The rules for using spaces are:

- Tokens cannot include spaces except if the token is a string literal or a delimited identifier.
- A token can be followed by a space.
- An ordinary token must be followed by a delimiter token or a space.
- If the syntax does not allow an ordinary token to be followed by a delimiter token, the ordinary token must be followed by a space.

## Uppercase and Lowercase

Any token can include lowercase letters. Lowercase letters in an ordinary token are always folded to uppercase. However, lowercase letters in a delimiter token are never folded to uppercase.

## Identifiers

An identifier is a token that is used to form a name. The two types of identifiers are host identifiers and SQL identifiers.

A host identifier is an identifier used to form the name of a host variable. Rules for forming host identifiers depend on the host language.

All other identifiers are SQL identifiers. SQL identifiers are of two types, ordinary and delimited.

## Ordinary SQL identifiers

An ordinary SQL identifier is a letter (the term "letter" includes the @, \$, and #) that may be followed by zero or more characters. Each character is a letter (including @, \$, #), digit, or the underscore character.

Ordinary SQL identifiers can include DBCS (Double-Byte Character Set) characters. The DBCS portions of the name must be delimited by the Shift-Out and Shift-In characters. The names are checked by SQL for paired shift characters. The maximum physical length of these names is the same as it was for prior versions of CA Datacom/DB. The Shift characters count in determining the length, and the DBCS characters take up two bytes each. Therefore you can get a maximum of 15 DBCS characters in a 32-byte identifier ((15 \* 2) for the DBCS characters + 2 for the shift characters). DBCS blanks (X'4040') are not allowed in the DBCS portion of an identifier.

**Note:** Do not use a reserved word as an ordinary SQL identifier. See the list of reserved words on [Reserved Words](#) (see page 43).

## Delimited SQL identifiers

A delimited SQL identifier is a sequence of one or more characters enclosed within SQL escape characters. Enclosing a name within SQL escape characters allows a name to be used which is the same as an SQL reserved word (see Reserved Words for a list of the SQL reserved words.) The escape character is the quotation mark (") unless the plan option STRDELIM= has been set to Q, in which case the escape character is the apostrophe (').

In the following example, the column named SELECT is enclosed in escape characters (quotation marks in the example) to show that the SQL reserved word SELECT is *not* what is meant.

```
CREATE TABLE SAMPLE
  ("SELECT" CHAR(6) DEFAULT 'SELECT',
  COL2 INTEGER)

...
SELECT "SELECT" FROM SAMPLE
WHERE "SELECT" = 'SELECT'
```

## Naming Conventions

The rules for forming a name depend on the type of object designated by the name. Syntax diagrams for SQL language elements and statements use different terms for different types of names. Following are terms used in the syntax diagrams.

### **accessor-id**

An SQL identifier that designates a user. Accessor IDs must be 1 to 18 characters.

### **authorization-id (AUTHID)**

An SQL identifier that designates a schema. Authorization IDs must be 1 to 18 characters.

The SQL name and the AUTHORIZATION occurrence name are the same and must be unique for all schemas.

### **column-name**

A name that designates a column of a table or view. The name can be unqualified or qualified. If you specify ANSI or FIPS for SQLMODE= in the SQL Preprocessor options, the unqualified form of a column name is an SQL identifier, 1 to 18 characters. For all other modes, the unqualified form of a column name can be 1 to 32 characters. The qualified form of a column name is a qualifier followed by a period and the column name. The qualifier is a table-name, a view-name or a correlation-name.

### **correlation-name**

An SQL identifier that designates a table, a view, or individual rows of a table or view. If you specify ANSI or FIPS for SQLMODE= in the SQL Preprocessor options, the correlation name can be 1 to 18 characters. For all other modes, the correlation name can be 1 to 32 characters. Also see [Correlation Names](#) (see page 515) and [SQL Index Binding](#) (see page 516).

### **cursor-name**

An SQL identifier that designates an SQL cursor. If you specify ANSI or FIPS for SQLMODE= in the SQL Preprocessor options, the cursor name can be 1 to 18 characters. For all other modes, the cursor name can be 1 to 32 characters.

### **descriptor-name**

A host identifier that designates an SQL Descriptor Area (SQLDA). The host identifier may be preceded by a colon. For more information on the SQLDA, see [SQL Descriptor Area \(SQLDA\)](#) (see page 869). For more information on host identifiers, see [Host Variables](#) (see page 520).

### **host-variable**

A sequence of tokens that designate one or more host variables. The host variable includes an identifier (see [Host Variables](#) (see page 520)).



**index-name**

A name that designates an index. The name can be qualified or unqualified. The unqualified index name in an SQL statement is implicitly qualified by the authorization ID of that statement. If you specify ANSI or FIPS for SQLMODE= in the SQL Preprocessor options, the unqualified form of an index name is an SQL identifier, 1 to 18 characters. For all other modes, the unqualified form of an index name can be 1 to 32 characters in length. The qualified form of an index name is an authorization ID followed by a period and the index name.

The SQL name of the index must be unique for all indexes owned by a specific schema (authorization ID). See [Naming the Index \(Key\)](#) (see page 416) for more information on naming an index.

**statement-name**

An identifier (1 to 18 characters) that designates a prepared SQL statement.

**procedure name**

The procedure name is the name that identifies a procedure. The name can be qualified with an authorization ID.

**SQL parameter name**

The SQL parameter name is the name of a parameter passed to an SQL Procedure (a LANGUAGE SQL procedure). When used in a SQL Procedure containing an SQL variable with a conflicting (matching) name, or a table or view reference where the table or view contains a conflicting column name, the name should be qualified using the procedure name.

**SQL variable name**

The SQL variable name is the name of a variable that is declared within a compound statement inside a SQL Procedure (a LANGUAGE SQL procedure). If the name conflicts with (matches) another SQL variable name (for example, from a nested compound statement such as a condition handler), an SQL parameter name, or a column contained within a referenced table or view, the SQL variable name should be qualified using the start-label of the compound statement that immediately contains it.

### **synonym**

An SQL identifier that designates a table or a view. A synonym can be used wherever a table-name or view-name can be used to reference a table or view. If you specify ANSI or FIPS for SQLMODE= in the SQL Preprocessor options, the synonym name is an SQL identifier, 1 to 18 characters. For all other modes, the synonym name can be 1 to 32 characters.

The SQL name of the synonym must be unique for all tables, views, and synonyms owned by a specific schema (authorization ID). See [Naming the Synonym](#) (see page 424) for more information on naming a synonym.

### **table-name**

A name that designates a table. The name can be qualified or unqualified. The unqualified table name in an SQL statement is implicitly qualified by the authorization ID of that statement. If you specify ANSI or FIPS for SQLMODE= in the SQL Preprocessor options, the unqualified form of a table name is an SQL identifier, 1 to 18 characters. For all other modes, the unqualified form of a table name can be 1 to 32 characters in length. The qualified form of a table name is an authorization ID followed by a period and the table name.

The SQL name of the table must be unique for all indexes, views and synonyms owned by a specific schema (authorization ID). See [Naming the Table](#) (see page 406) for more information on naming a table.

### **view-name**

A name that designates a view. The name can be qualified or unqualified. The unqualified view name in an SQL statement is implicitly qualified by the authorization ID of that statement. If you specify ANSI or FIPS for SQLMODE= in the SQL Preprocessor options, the unqualified form of a view name is an SQL identifier, 1 to 18 characters. For all other modes, the unqualified form of a view name can be 1 to 32 characters. The qualified form of a view name is an authorization ID followed by a period and the view name.

The SQL name of the view must be unique for all tables, views and synonyms owned by a specific schema (authorization ID). See [Naming the View](#) (see page 419) for more information on naming a view.

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

When you create SQL tables, views and synonyms, the SQL name is prefixed by the authorization ID to create the TABLE, VIEW and SYNONYM occurrence name. The format is *authid-sqlname*.

Together, the authorization ID and SQL name of each table, view and synonym must be unique within the schema. For example, the names JONES.DEPTTBL (for a table) and JONES.DEPTTBL (for a view) **are not** unique since both are owned by the JONES schema, but the names JONES.DEPTTBL (for a table) and SMITH.DEPTTBL (for a view) **are** unique because they are owned by different schemas.

## Authorization ID

An authorization ID is a string of 1 to 18 characters. This rule applies for both ANSI mode and the CA Datacom/DB extended mode.

Authorization IDs are used to identify schemas. For example, to select from a table named PAY in the schema named CA, the table name is preceded by the authorization ID:

```
SELECT *  
FROM CA.PAY
```

**Note:** Do not specify authorization IDs for tables and views that are to be used in Dynamic Plan Selection. (In a z/OS environment, Dynamic Plan Selection online requires CA Datacom CICS Services Release 2.5 and higher. Dynamic Plan Selection is batch only in z/VSE.) For more information on Dynamic Plan Selection, see the *CA Datacom/DB Database and System Administration Guide*.

## Values

The smallest unit of data that can be manipulated in SQL is called a value. How a value is interpreted depends on the data type of the source. Sources of values include:

- [Scalar Functions](#) (see page 554)
- [Expressions](#) (see page 527)
- [Special Registers](#) (see page 533)
- [Labeled Duration](#) (see page 535)
- [Literals](#) (see page 510)
- [Column Names](#) (see page 514)
- [Host Variables](#) (see page 520)
- [Indicator Variables](#) (see page 524).
- [SQL Parameters](#) (see page 525).
- [SQL Variables.](#) (see page 525)

## Data Types

The valid SQL data types are:

- CHARACTER
- VARCHAR
- LONG VARCHAR
- FLOAT
- NUMERIC
- DECIMAL
- INTEGER
- SMALLINT
- REAL
- DOUBLE PRECISION
- DATE
- TIME
- TIMESTAMP
- GRAPHIC
- VARGRAPHIC
- LONG VARGRAPHIC

## DATE, TIME, and TIMESTAMP

DATE, TIME, and TIMESTAMP are special SQL data types that are stored and manipulated in CA Datacom/DB as BINARY data with lengths 4, 3, and 10 respectively. However, when SQL retrieves this type of data from the database, it sends it back to the user in CHARACTER form. When you specify this type of data to send to the database, you specify it in CHARACTER form. See [Character String Literals](#) (see page 510).

If you are accessing the data with non-SQL commands, the data is in the internal form which is BINARY and you must perform the conversion from the internal form yourself. 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.

## Host Variable Data Types

Information about ANSI standard and non-standard host variable data types can be found in [Character Strings](#) (see page 495) for COBOL and in Host-Variable Declarations for PL/I. Host variable declarations for Assembler can be found in Host Variable Declarations for Assembler. Host variable declarations for the C language are described in Rules for Coding Host-Variables in C.

## SQL Data Types

See [Data Types](#) (see page 485) for the valid SQL data types syntax diagram.

## CA Datacom/DB Data Types

When defining a column, specify its data type in the TYPE attribute of the FIELD entity-occurrence or in the SQL statement. Refer to the *CA Datacom Datadictionary User Guide* and the *CA Datacom/DB SQL User Guide* for details. CA Datacom/DB supports the following data types:

### **Binary, halfword - 2 bytes**

CA Datacom Datadictionary Valid Entry: B

SQL Data Type: SMALLINT

CA Ideal Support: Supported

### **Binary, fullword - 4 bytes**

CA Datacom Datadictionary Valid Entry: B

SQL Data Type: INTEGER or INT

CA Ideal Support: Supported

### **Binary, length = 4, SEMANTIC-TYPE=SQL-DATE**

CA Datacom Datadictionary Valid Entry: B

SQL Data Type: DATE

CA Ideal Support: Supported

### **Binary, length = 3, SEMANTIC-TYPE=SQL-TIME**

CA Datacom Datadictionary Valid Entry: B

SQL Data Type: TIME

CA Ideal Support: Supported

**Binary, length = 10, SEMANTIC-TYPE=SQL-STMP**

CA Datacom Datadictionary Valid Entry: B

SQL Data Type: TIMESTAMP

CA Ideal Support: Supported

**Character**

CA Datacom Datadictionary Valid Entry: C

SQL Data Type: CHARACTER or CHAR

CA Ideal Support: Supported

**Character data type with mixed DBCS and SBCS with SEMANTIC-TYPE=MIXED**

CA Datacom Datadictionary Valid Entry: C

SQL Data Type: CHARACTER with FOR MIXED DATA

CA Ideal Support: Supported as character

**Varying-length character string of length 1 to maximum row size**

CA Datacom Datadictionary Valid Entry: V

SQL Data Type: VARCHAR

CA Ideal Support: Supported

**V data type with mixed DBCS and SBCS with SEMANTIC-TYPE=MIXED**

CA Datacom Datadictionary Valid Entry: V

SQL Data Type: VARCHAR

CA Ideal Support: Supported

**V data type with attribute LONG=Y**

Varying-length character string whose maximum length is determined by the amount of space available in a block after all fixed-length fields have been subtracted from the blocksize (assumes one record per block). See following notes.

CA Datacom Datadictionary Valid Entry: V

SQL Data Type: LONG VARCHAR

CA Ideal Support: Supported

**Packed decimal**

CA Datacom Datadictionary Valid Entry: D

SQL Data Type: DECIMAL or DEC, maximum length = 16 bytes

CA Ideal Support: Supported

**Numeric, zoned decimal**

CA Datacom Datadictionary Valid Entry: N

SQL Data Type: NUMERIC, maximum length = 31 bytes

CA Ideal Support: Supported

**Hexadecimal, two-byte hexadecimal display**

CA Datacom Datadictionary Valid Entry: H

SQL Data Type: Not supported.

CA Ideal Support: Not supported

**Short floating-point, fullword aligned**

CA Datacom Datadictionary Valid Entry: S

SQL Data Type: CHAR

CA Ideal Support: Not supported

**Long floating-point, doubleword aligned**

CA Datacom Datadictionary Valid Entry: L

SQL Data Type: FLOAT, REAL, DOUBLE PRECISION

CA Ideal Support: Not supported

**Extended floating-point, 16-byte aligned**

CA Datacom Datadictionary Valid Entry: E

SQL Data Type: CHAR

CA Ideal Support: Not supported

**Double-byte character set (DBCS)**

See notes following this list.

CA Datacom Datadictionary Valid Entry: Y

SQL Data Type: GRAPHIC

CA Ideal Support: Not supported

**Mixed DBCS and single byte**

See notes following this list.

CA Datacom Datadictionary Valid Entry: Z

SQL Data Type: CHAR with FOR MIXED DATA

CA Ideal Support: Supported as character



**Graphics data**

CA Datacom Datadictionary Valid Entry: G

SQL Data Type: GRAPHIC

CA Ideal Support: Not supported

**Varying-length double-byte character set of length 2 to maximum row size**

CA Datacom Datadictionary Valid Entry: W

SQL Data Type: VARGRAPHIC

CA Ideal Support: Not supported

**Varying-length double-byte character set of length 2 to maximum row size**

With attribute LONG=Y varying-length double-byte character set whose maximum length is determined by the amount of space available in a block after all fixed length fields have been subtracted from the blocksize (assumes one record per block). See the notes following this list.

CA Datacom Datadictionary Valid Entry: W

SQL Data Type: LONG VARGRAPHIC

CA Ideal Support: Not supported

**Kanji, same as Y and G**

See notes following this list.

CA Datacom Datadictionary Valid Entry: K

SQL Data Type: GRAPHIC

CA Ideal Support: Not supported

**PL/I bit representation**

CA Datacom Datadictionary Valid Entry: T

SQL Data Type: Not supported

CA Ideal Support: Not supported

**Halfword binary, aligned**

CA Datacom Datadictionary Valid Entry: 2

SQL Data Type: SMALLINT

CA Ideal Support: Not supported

**Fullword binary, aligned**

CA Datacom Datadictionary Valid Entry: 4

SQL Data Type: INTEGER or INT

CA Ideal Support: Not supported

**Doubleword binary, aligned**

CA Datacom Datadictionary Valid Entry: 8

SQL Data Type: Not supported

CA Ideal Support: Not supported

**Notes:**

- In CA Datacom Datadictionary the data types K, Y, and Z are being replaced by GRAPHIC and CHARACTER data types. You can use the DDCNVLM utility to update FIELD entity-occurrences with data types of K or Y to G (GRAPHIC) and data type Z to C (CHARACTER) with SEMANTIC-TYPE=MIXED. See the *CA Datacom Installation Guide* for more information about DDCNVLM.
- SMALLINT and INTEGER data types do not allow decimals.
- Aligned field types are aligned in records or tables, not in elements and keys.
- Those lengths or types not specified as an SQL data type are treated as character through SQL.
- VARCHAR (TYPE=V) and VARGRAPHIC (TYPE=W) have a two-byte binary length prefix added to the value you specify for the LENGTH attribute.

## SQL Data Type Support for All CA Datacom/DB Tables

Tables created with prior versions of CA Datacom/DB may contain data types unrecognizable by SQL. For data to be retrieved from those tables, unrecognizable data types must be viewed as recognizable data types.

The chart on the following page shows the SQL-supported values for four CA Datacom Datadictionary FIELD attributes (TYPE, JUSTIFICATION, SIGN, and TYPE-NUMERIC), their meaning, and the equivalent SQL data type. When retrieving data from any column (field) that is defined in CA Datacom Datadictionary with a combination of attribute-values not included in this chart, CA Datacom/DB returns the data to an SQL application as if it were CHARACTER. That is, CA Datacom/DB presents the data without any translation or interpretation.

The **CA Datacom Datadictionary FIELD Attributes** have the following meanings:

**TYPE**

Represents CA Datacom Datadictionary FIELD Attributes **TYPE**

**JUST**

Represents CA Datacom Datadictionary FIELD Attributes **JUSTIFICATION**

**SIGN**

Represents CA Datacom Datadictionary FIELD Attributes **SIGN**

**T-N**

Represents CA Datacom Datadictionary FIELD Attributes **TYPE-NUMERIC**

**Binary, halfword - 2 bytes**

CA Datacom Datadictionary Valid Entry: B

SQL Data Type: SMALLINT

CA Ideal Support: Supported

**Binary, fullword - 4 bytes**

CA Datacom Datadictionary Valid Entry: B

SQL Data Type: INTEGER or INT

CA Ideal Support: Supported

**Binary, length = 4, SEMANTIC-TYPE=SQL-DATE**

CA Datacom Datadictionary Valid Entry: B

SQL Data Type: DATE

CA Ideal Support: Supported

**Binary, length = 3, SEMANTIC-TYPE=SQL-TIME**

CA Datacom Datadictionary Valid Entry: B

SQL Data Type: TIME

CA Ideal Support: Supported

**Binary, length = 10, SEMANTIC-TYPE=SQL-STMP**

CA Datacom Datadictionary Valid Entry: B

SQL Data Type: TIMESTAMP

CA Ideal Support: Supported

**Character**

CA Datacom Datadictionary Valid Entry: C

SQL Data Type: CHARACTER or CHAR

CA Ideal Support: Supported

**Character data type with mixed DBCS and SBCS with SEMANTIC-TYPE=MIXED**

CA Datacom Datadictionary Valid Entry: C

SQL Data Type: CHARACTER with FOR MIXED DATA

CA Ideal Support: Supported as character

**Varying-length character string of length 1 to maximum row size**

CA Datacom Datadictionary Valid Entry: V

SQL Data Type: VARCHAR

CA Ideal Support: Supported

**V data type with mixed DBCS and SBCS with SEMANTIC-TYPE=MIXED**

CA Datacom Datadictionary Valid Entry: V

SQL Data Type: VARCHAR

CA Ideal Support: Supported

**V data type with attribute LONG=Y**

Varying-length character string whose maximum length is determined by the amount of space available in a block after all fixed-length fields have been subtracted from the blocksize (assumes one record per block). See following notes.

CA Datacom Datadictionary Valid Entry: V

SQL Data Type: LONG VARCHAR

CA Ideal Support: Supported

**Packed decimal**

CA Datacom Datadictionary Valid Entry: D

SQL Data Type: DECIMAL or DEC, maximum length = 16 bytes

CA Ideal Support: Supported

**Numeric, zoned decimal**

CA Datacom Datadictionary Valid Entry: N

SQL Data Type: NUMERIC, maximum length = 31 bytes

CA Ideal Support: Supported

**Hexadecimal, two-byte hexadecimal display**

CA Datacom Datadictionary Valid Entry: H

SQL Data Type: n/a

CA Ideal Support: Not supported

**Short floating-point, fullword aligned**

CA Datacom Datadictionary Valid Entry: S

SQL Data Type: CHAR

CA Ideal Support: Not supported

**Long floating-point, doubleword aligned**

CA Datacom Datadictionary Valid Entry: L  
SQL Data Type: FLOAT, REAL, DOUBLE PRECISION  
CA Ideal Support: Not supported

**Extended floating-point, 16-byte aligned**

CA Datacom Datadictionary Valid Entry: E  
SQL Data Type: CHAR  
CA Ideal Support: Not supported

**Double-byte character set (DBCS)**

See notes following this list.  
CA Datacom Datadictionary Valid Entry: Y  
SQL Data Type: GRAPHIC  
CA Ideal Support: Not supported

**Mixed DBCS and single byte**

See notes following this list.  
CA Datacom Datadictionary Valid Entry: Z  
SQL Data Type: CHAR with FOR MIXED DATA  
CA Ideal Support: Supported as character

**Graphics data**

CA Datacom Datadictionary Valid Entry: G  
SQL Data Type: GRAPHIC  
CA Ideal Support: Not supported

**Varying-length double-byte character set of length 2 to maximum row size**

CA Datacom Datadictionary Valid Entry: W  
SQL Data Type: VARGRAPHIC  
CA Ideal Support: Not supported

**Varying-length double-byte character set of length 2 to maximum row size**

With attribute LONG=Y varying-length double-byte character set whose maximum length is determined by the amount of space available in a block after all fixed length fields have been subtracted from the blocksize (assumes one record per block). See the following notes.

CA Datacom Datadictionary Valid Entry: W

SQL Data Type: LONG VARGRAPHIC

CA Ideal Support: Not supported

**Kanji, same as Y and G**

See notes following this list.

CA Datacom Datadictionary Valid Entry: K

SQL Data Type: GRAPHIC

CA Ideal Support: Not supported

**PL/I bit representation**

CA Datacom Datadictionary Valid Entry: T

SQL Data Type: n/a

CA Ideal Support: Not supported

**Halfword binary, aligned**

CA Datacom Datadictionary Valid Entry: 2

SQL Data Type: SMALLINT

CA Ideal Support: Not supported

**Fullword binary, aligned**

CA Datacom Datadictionary Valid Entry: 4

SQL Data Type: INTEGER or INT

CA Ideal Support: Not supported

**Doubleword binary, aligned**

CA Datacom Datadictionary Valid Entry: 8

SQL Data Type: CHAR(8)

CA Ideal Support: Not supported

**Notes:**

- In CA Datacom Datadictionary the data types K, Y, and Z are being replaced by GRAPHIC and CHARACTER data types. You can use the DDCNVLM utility to update FIELD entity-occurrences with data types of K or Y to G (GRAPHIC) and data type Z to C (CHARACTER) with SEMANTIC-TYPE=MIXED. See the *CA Datacom Installation Guide* for more information about DDCNVLM.
- SMALLINT and INTEGER data types do not allow decimals.
- Aligned field types are aligned in records or tables, not in elements and keys.
- Those lengths or types not specified as an SQL data type are treated as character through SQL.
- VARCHAR (TYPE=V) and VARGRAPHIC (TYPE=W) have a two-byte binary length prefix added to the value you specify for the LENGTH attribute.

## Character Strings

A character string is a sequence of any EBCDIC characters. The length of the string is determined by the length attribute of the column. The data type for a character string is CHARACTER. The default value for the CHARACTER data type length is one byte. Also see [Data Types](#) (see page 485).

In extended mode you can use character strings to represent dates, times, and timestamps. See [Character String Literals](#) (see page 510) for more information.

## VARCHAR and LONG VARCHAR

The VARCHAR and LONG VARCHAR data types provide for varying-length character strings.

### **VARCHAR**

VARCHAR(n) specifies a varying-length character string of maximum length n. The length may range from 1 to the maximum row size.

The physical size in bytes of a VARCHAR(n) column is:

$$n + 2 + x$$

where x=1 if the column allows nulls, else x=0.

The 2 is added because of the two-byte current-length field which precedes the character string. The 1 is added if the column allows nulls because of the one-byte null indicator flag which precedes the length and data fields.

## LONG VARCHAR

Specifies a varying-length character string whose maximum length is determined by the amount of space available in a block.

The physical size of a LONG VARCHAR field varies with the block size of the table and the number of LONG VARCHAR fields defined for the table. Assuming:

m = Maximum row size (block size - 14)

i = The sum of the byte counts of all columns in the table that are not LONG VARCHAR

j = the number of LONG VARCHAR columns in the table

k = the number of LONG VARCHAR columns which allow nulls

then the physical size in bytes of a LONG VARCHAR column is:

$$2 * (\text{INTEGER} ((\text{INTEGER} ((m - i - k) / j)) / 2))$$

LONG VARCHAR columns get whatever space is left in the block after all of the other columns are defined. If only one LONG VARCHAR is defined, it gets all of the space left in the block that is not taken up by the other columns. If there is more than one LONG VARCHAR column, they share the space evenly. Defining any LONG VARCHAR columns causes the row size of the table to be equal to the block size (minus block overhead).

Defining a LONG VARCHAR column is exactly equivalent to defining a VARCHAR(n) where n = the size as calculated above. If the block size of the table is later changed, the size of its LONG VARCHAR columns is not recalculated. When modifying a LONG VARCHAR column using ALTER TABLE, the column is treated like any VARCHAR(n) column. The LONG VARCHAR data type is provided to simplify defining a column that is as long as possible to be used for storing text, binary data, and so forth.

### Using VARCHAR Columns

Each VARCHAR or LONG VARCHAR column value is made up of two parts, a SMALLINT current-length subfield, which contains the length in bytes of the character string, followed by the character string itself. The length specified when the column is defined is the maximum length; at any given time, a column value may contain a character string whose length is between 0 and the maximum length of the column.

In a COBOL program, a VARCHAR host variable must have the following form:

```
01 VARI.  
   49 VARI-LEN      PIC S9(4) USAGE COMP.  
   49 VARI-TEXT     PIC X(n).
```



Where VAR1, VAR1-LEN, and VAR1-TEXT are user-defined names, and n is the maximum length for the VARCHAR column. The group-level may be numbered 01 through 48. The group must contain two elementary items with the level number of 49. The first elementary item must be a SMALLINT integer variable. The second elementary item must have the same description as a fixed-length character string. When referring to the VARCHAR host variable in an embedded SQL statement, the group name is used, for example :VAR1 in the following:

```
INSERT INTO TABLE1 VALUES (:VAR1, 50)
```

Before the INSERT statement is executed, the character string to insert must be moved to VAR1-TEXT, and VAR1-LEN must be set to the length of the data in VAR1-TEXT which is to be inserted into the column.

When a VARCHAR column value is retrieved, CA Datacom/DB fills in the length as well as the text. For example, after the following statement is executed:

```
FETCH CURSOR1 INTO :VAR1
```

VAR1-LEN is set to the length of the character data returned in VAR1-TEXT.

Null indicators work as they do with any other data type. For instance, to insert a NULL value into a column, define a null indicator variable, set it to -1, and specify the null indicator variable and the variable for the column value:

```
01 VAR1.
  49 VAR1-LEN      PIC S9(4) USAGE COMP.
  49 VAR1-TEXT     PIC X(n).
01 VAR1-NI        PIC S9(4) USAGE COMP.
SET VAR1-NI = TO -1.
INSERT INTO TABLE1 VALUES (:VAR1:VAR1-NI, 500)
```

In this case, neither VAR1-TEXT nor VAR1-LEN need be set to any particular value, because the null indicator is checked first. Similarly, when retrieving a VARCHAR value that might be null, provide a null indicator value and check the null indicator first. If the null indicator indicates that the value is null, the values in VAR1-LEN and VAR1-TEXT are undefined.

```
FETCH CURSOR1 INTO :VAR1:VAR1-NI

IF VAR1-NI = -1 THEN
  PERFORM 'STRING-OUT' USING 4 'NULL'
ELSE
  PERFORM 'STRING-OUT' USING VAR1-LEN VAR1-TEXT
END-IF
```

### Various Rules Related to VARCHAR

In general, you can use a varying-length character string column or variable anywhere a fixed-length character string or column could be used.

When a string of length *n* is assigned to a varying-length string variable with a maximum length greater than *n*, the characters after the *n*th character of the variable are undefined and may or may not be set to blanks.

Varying-length strings that differ only in the number of trailing blanks are considered equal.

In CA Datacom/DB, character string constants are considered to be fixed-length character strings and are limited to 32720 characters.

### MIXED Data

Three semantic types are allowed on CHAR, VARCHAR, and LONG VARCHAR columns: FOR MIXED DATA, FOR SBCS DATA, and FOR BIT DATA.

FOR MIXED DATA means that Double-Byte Character Set (DBCS) characters are allowed in values stored in the column, in addition to EBCDIC (Single-Byte Character Set (SBCS)) characters. This is relevant when SQL is processing a value in the column, since it must recognize the Shift-Out and Shift-In characters that delimit DBCS substrings. FOR SBCS DATA means that DBCS characters are not used in the column. FOR BIT DATA means that the data is a string of binary data rather than a string of characters.

If a semantic type is not specified when the column is created, the default is the semantic type that was specified on the CXXMAINT OPTION=ALTER,DBCS=xxx option. If xxx was IS or FS, the default is FOR SBCS DATA. If xxx was IM or FM, the default is FOR MIXED DATA.

Default values for MIXED columns may include DBCS characters (each sequence of DBCS characters must be delimited by Shift characters).

**Note:** SQL does not check to make sure Shift characters are not used in SBCS strings, and there is no validation of DBCS characters. X'42' is used as the first byte of any (non-blank) DBCS character generated by SQL.

## GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC

The GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC data types provide Double-Byte Character Set (DBCS) support. Each character is two bytes long. The precision of one of these columns is the maximum number of two-byte characters that can be stored, not the physical length.

DBCS characters in SQL statements are delimited by the Shift-Out and Shift-In characters. Shift characters are either the IBM-defined characters (X'0E' and X'0F') or the Fujitsu-defined characters (X'28' and X'29'), as specified in the CXXMAINT OPTION=ALTER,DBCS=xxx option. If xxx is IS or IM, the shift characters are the IBM-defined characters. If xxx is FS or FM, the shift characters are the Fujitsu-defined characters.

**Note:** SQL does not check to make sure Shift characters are not used in SBCS strings, and there is no validation of DBCS characters. X'42' is used as the first byte of any (non-blank) DBCS character generated by SQL.

Default values for GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC columns are expressed as graphic literals in either the COBOL or PL/I formats. In these default literals the quote must always be the single-quote. The existing limit of 20 bytes maximum for all default values also applies to the GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC data types.

### Ordering on DBCS Columns

GRAPHIC and VARGRAPHIC columns are indexed and/or used as sort fields, but the ordering is strictly byte-for-byte. There is no support provided for ordering the DBCS columns in other ways.

Indexing and ordering on MIXED columns is also strictly byte-for-byte. There is no normalizing of the strings so that equivalent DBCS versions of SBCS strings or substrings would compare equal.

### Katakana Support

The code points used for the encoding of the Katakana language overlap the code points assigned to lower case EBCDIC letters, so special processing is required to support Katakana. If the CXXMAINT ALTER LANGUAGE option has been turned on, SQL supports Katakana, which also means lower-case English letters are not available. Even when LANGUAGE is K, SQL uses X'42', not X'43' as the first byte of any (non-blank) DBCS character it generates.

## Numeric Data Types

The numeric data types can be defined in all host languages.

All binary integers and decimal numbers have a sign and a precision. The precision is the total number of binary or decimal digits, excluding the sign.

### **NUMERIC**

A zoned decimal number with an implicit decimal point. The position of the decimal point determined by the precision and scale of the number.

The precision is the total number of decimal digits. The maximum precision is 31 digits.

The scale is the number of digits in the fractional part of the number. The scale cannot be negative and it cannot be greater than the precision.

All values of a decimal column have the same precision and scale. The range of a decimal variable or the numbers in a decimal column is -n to n. The absolute value of n is the largest number that can be represented with the applicable precision and scale.

### **DECIMAL**

A packed decimal number with an implicit decimal point. The position of the decimal point is determined by the precision and scale of the number.

The precision is the total number of decimal digits. The maximum precision is 31 digits.

The scale is the number of digits in the fractional part of the number. The scale cannot be negative and it cannot be greater than the precision.

All values of a decimal column have the same precision and scale. The range of a decimal variable or the numbers in a decimal column is -n to n. The absolute value of n is the largest number that can be represented with the applicable precision and scale.

### **SMALLINT**

A binary integer with a precision of 15 bits. The range of small integers is -32768 to 32767.

### **INTEGER**

A binary integer with a precision of 31 bits. The range of large integers is -2147483648 to 2147483647.

### **FLOAT**

A long (64 bits) floating-point number. The range of magnitudes is approximately 5.4E-79 to 7.2E+75.

**REAL**

A long (64 bits) floating-point number. The range of magnitudes is approximately 5.4E-79 to 7.2E+75.

**DOUBLE PRECISION**

A long (64 bits) floating-point number. The range of magnitudes is approximately 5.4E-79 to 7.2E+75.

## Basic Operations (Assignment and Comparison)

The basic operations of the SQL language are assignment and comparison.

Assignment operations are performed during the execution of:

- INSERT
- UPDATE
- FETCH

Comparison operations are performed during the execution of statements that include predicates and other language elements such as:

- DISTINCT
- GROUP BY
- MAX
- MIN
- ORDER BY

The basic rule for both operations is that numbers and strings are not compatible. This means that:

- Numbers and strings cannot be compared.
- Numbers cannot be assigned to string columns or variables.
- Strings cannot be assigned to numeric columns or variables.

All character strings are compatible. For example one character string can be compared to another character string.

All numbers are compatible. (See [Numeric Assignments](#) (see page 502) and [Numeric Comparisons](#) (see page 507) for information on conversions made during assignment and comparison operations.)

## Numeric Assignments

The following table lists the conversion rules which apply when assigning a number of one data type to another data type.

Conversion implies that the value is represented in a form compatible for the assignment operation.

To determine the rule which applies when assigning a number of one data type to another data type, locate the original data type in the left column. Next, locate the target data type in the vertical columns to the right.

The rule number specified at the intersection of the two columns is the rule which applies during the assignment operation. See the list of numbered rules following the table for an explanation of the conversion required during the assignment operation.

Applicable Conversion Rules When Assigning Data Types														
Original Data Type:	Target Data Type:													
	F	R	DP	D	SI	LI	N	V	C	DT	T	TS	G	VG
F				5	2	2	5							
R				5	2	2	5							
DP				5	2	2	5							
D	1	1	1	3	6	6	3							
SI	1	1	1	4			4							
LI	1	1	1	4			4							
N	1	1	1	3	6	6	3							
V									8					
C								9		7	7	7		
DT									7	7				
T									7	7				
TS									7			7		
G													10	10
VG													10	10

The following list contains explanations of the numbered rules referenced in the previous table.

1. **Decimal or integer to floating-point**

Floating-point numbers are approximations of real numbers. Thus, decimal and integer numbers may not be identical to the original number when assignment is to a floating-point column or variable.

2. **Floating-point to integer**

The fractional part of the number is lost on this assignment.

3. **Decimal to decimal**

When a decimal number is assigned to a decimal column or variable, the number is converted, if necessary, to the precision and scale of the target.

In the whole part of the number, the necessary number of leading zeros is appended or eliminated.

In the fractional part of the number, the necessary number of leading zeros is appended, or the necessary number of digits exceeding the scale is eliminated.

4. **Integer to decimal**

When an integer is assigned to a decimal column or variable, the number is converted to the DECIMAL data type first. Next, it is converted to the precision and scale of the target. For example if X'0005' is placed in a DECIMAL with precision equal to 3 and scale equal to 2, it appears as pack decimal '500C'.

The precision of a small integer, when converted to DECIMAL, is 5, and the scale is 0.

The precision of a large integer, when converted to DECIMAL, is 11, and the scale is 0.

5. **Floating-point to decimal**

When a floating-point number is assigned to DECIMAL or NUMERIC, the number is first converted to a temporary packed decimal number of precision 31. If necessary, the number is then truncated to the precision and scale of the target.

In conversion, the number is rounded (using floating-point arithmetic) to a precision of 31 decimal digits. If the representation requires more than 31 digits to the left of the decimal point, an error is reported. Otherwise, the scale is given the largest possible value that allows the whole part of the number to be represented without loss of significance.

**6. Decimal to integer**

The fractional part of the decimal number is truncated.

The maximum range of values which can be represented for each data type is:

<b>Data Type</b>	<b>Minimum Value</b>	<b>Maximum Value</b>
SMALLINT	-32768	32767
INTEGER	-2147483648	2147483647
DECIMAL	(31 digits)	(31 digits)
NUMERIC	(31 digits)	(31 digits)

**7. DATE, TIME, and TIMESTAMP**

See "Assignment for Dates, Times, and Timestamps" on [Assignment for Dates, Times, and Timestamps](#) (see page 506).

**8. VARCHAR to CHAR**

If the varying-length string value is shorter than the length attribute of the CHAR column or variable, the string is padded on the right with blanks until it is the required length.

If the varying-length string value is longer than the length attribute of a CHAR column, an error occurs.

If the varying-length string value is longer than the length attribute of a CHAR variable, the string is truncated on the right by the required number of characters (a 'W' is assigned to SQLWARN1 of the SQLCA if this occurs).

**9. CHAR to VARCHAR**

If the length attribute of the CHAR column or variable is less than the maximum length of the VARCHAR column or variable, the CHAR string is copied and the length is set to the length of the CHAR string.

If the length attribute of the CHAR column or variable is longer than the maximum length of the VARCHAR target, an error occurs when the target is a column, or truncation occurs when the target is a variable.



#### 10. MIXED Data

When a GRAPHIC value is assigned to a larger-precision target, the value is padded with DBCS blanks (X'4040'). When MIXED data is enabled via the CXXMAINT option, SQL does special processing as follows:

- When a MIXED string is moved to a smaller-precision MIXED host variable, SQL ensures that the truncation results in a well-formed string. If there is an unpaired Shift-Out in the result:
  - If the last byte in the truncated string is the first byte of a DBCS character, that last byte is overlaid with a Shift-In.
  - If the last byte in the truncated string is the second byte of a DBCS character, the last DBCS character is replaced by a Shift-In and an SBCS blank.
  - If the last byte of the truncated string is a Shift-Out, it is overlaid with an SBCS blank.
  - If the next-to-last byte of the truncated string is a Shift-Out, it and the last byte are overlaid with SBCS blanks.
- SQL issues an error if there are unpaired Shift codes in a string when assigning it to a column or host variable.

## String Assignment

The basic rule of string assignment is that the length of a string assigned to a column must not be greater than the length attribute of the column. Trailing blanks are included in the length of the string.

When a string is assigned to a fixed-length string column or variable and the length of the string is less than the length attribute of the target, the string is padded on the right with the necessary number of blanks.

When a string is assigned to a variable and the string is longer than the length attribute of the variable, the string is truncated on the right by the necessary number of characters.

## Assignment for Dates, Times, and Timestamps

The basic rule for date, time, and timestamp assignments is that a value may only be assigned to a column with a matching data type, or to a fixed or varying length character string variable or column.

If the assignment of a date or time value is made to a character string variable or column, conversion to a string representation is automatic. Leading zeros are not omitted from any part of the date, time or timestamp. The required length of the target varies depending on the format of the string representation. If the length of the target is greater than required, it is padded on the right with blanks. If the length of the target is less than required, the result depends on the type of date or time value involved, as well as on the type of target. If the target is a column, truncation is not allowed. The length must therefore be 10 for a date, 8 for a time, and 26 for a timestamp.

If the target is a host variable, the following rules apply:

### **Date Assignments**

If the string or column length is less than ten bytes, an error occurs.

### **Time Assignments**

If the USA format is used (see [Character String Literals](#) (see page 510) for additional format information), the length of your host variable or string column must not be less than 8. Otherwise, its length must not be less than 5.

If ISO or JIS formats are used, and if the length of the host variable is less than 8, the seconds part of the time is omitted from your result and assigned to your indicator variable. The SQLCA-WARNING(2) field of the SQLCA (SQLWARN1 in the DB2 format) is set to alert you to the omission.

### **Timestamp Assignments**

If the string or column length is less than 19 bytes, an error occurs. If the length is less than 26 bytes but greater than or equal to 19 bytes, trailing digits of the microseconds part of the value are omitted.

## Numeric Comparisons

Numbers are compared according to their algebraic value. In some cases, one of the operands must be converted to another data type before the comparison operation is performed.

Conversion implies that each of the two values are represented in a form compatible for the comparison operation.

This conversion is performed automatically when the specified operands are of different data types. General rules for conversion during a comparison operation are:

- Integer numbers are compared to decimal numbers by converting the integer number to DECIMAL (packed decimal).
- Decimal and integer numbers are compared to floating-point numbers by converting the decimal or integer number to FLOAT.

The following table lists the conversions that must take place before two numbers can be compared.

To determine the data type used in the comparison operation, locate the data type of the first operand in the left column. Next, locate the data type of the second operand in the vertical columns to the right.

The data type specified at the intersection of the two columns is the data type used in the actual comparison operation. In some cases, this data type is different from that of both operands, indicating that both operands are converted before the comparison is made.

Conversions When Comparing Data Types														
	Second Operand:													
First Operand:	F	R	DP	D	SI	LI	N	V	C	DT	T	TS	G	VG
F	X	F2	F2	F2	F2	F2	F2	●	●	●	●	●	●	●
R	F2	X	F2	F2	F2	F2	F2	●	●	●	●	●	●	●
DP	F2	F2	X	F2	F2	F2	F2	●	●	●	●	●	●	●
D	F	F	F	X <sup>1</sup>	D	D	D <sup>1</sup>	●	●	●	●	●	●	●
SI	F	F	F	D	X	LI	D	●	●	●	●	●	●	●
LI	F	F	F	D	LI	X	D	●	●	●	●	●	●	●
N	F	F	F	D <sup>1</sup>	D	D	D <sup>1</sup>	●	●	●	●	●	●	●
V	●	●	●	●	●	●	●	X	X	●	●	●	●	●
C	●	●	●	●	●	●	●	X	X	DT <sup>3</sup>	T <sup>3</sup>	TS <sup>3</sup>	●	●
DT	●	●	●	●	●	●	●	●	●	X	●	●	●	●
T	●	●	●	●	●	●	●	●	●	●	X	●	●	●
TS	●	●	●	●	●	●	●	●	●	●	●	X	●	●
G	●	●	●	●	●	●	●	●	●	●	●	●	X <sup>4</sup>	X <sup>4</sup>
VG	●	●	●	●	●	●	●	●	●	●	●	●	X <sup>4</sup>	X <sup>4</sup>

**Note:** Be aware that there is one exception to the information in this table. Normally, there is no comparison of any type of a number with character data, but an equal (=) comparison is allowed between unsigned numeric (zoned decimal) data and character data of the same length for joins between tables. Because this is also allowed when both operands are in the same table, a join does not have to be involved, but the exception was made to facilitate joining tables designed at different times. The exception works for equal comparisons of the same length by treating the unsigned zone field as a character column for comparison purposes.

In the previous tables, the following numbered rules are referenced:

**1.**

For decimal numbers, the comparison is done in DECIMAL form. The number with the smallest scale is extended with zeros to match the scale of the other number before the comparison.

**2.**

Any number compared to a floating-point number must be converted to FLOAT. Two floating-point numbers are equal only if the bit configurations of their normalized form are identical.

**3.**

See the information about dates, times, and timestamps on [Comparisons for Dates, Times, and Timestamps](#) (see page 510).

**4.**

MIXED Data Considerations: In SQL, string comparisons are strictly byte-for-byte, so predicates involving MIXED strings may not give the desired results. For instance, a string with the SBCS version of XYZ does not compare as equal to a string with a Shift-Out, then the DBCS version of XYZ, then a Shift-In. You can use the VARGRAPHIC scalar function to normalize both strings into VARGRAPHIC data types before doing the comparison.

## String Comparisons

The comparison of two strings is determined by the comparison of the corresponding bytes of each string.

If the strings are not the same length, the comparison is made with a temporary copy of the shorter string. The temporary copy is padded on the right with blanks so that it has the same length as the longest string.

Two strings are equal if all corresponding bytes are equal.

If two strings are not equal, their relation is determined by the comparison of the first pair of unequal bytes from the left end of the strings. This comparison is made according to the EBCDIC collating sequence.

## Comparisons for Dates, Times, and Timestamps

A value for a date, time, or timestamp can be compared either with another value of the same data type, or with a string representation of that data type. All comparisons are chronological, that is to say, the farther a point in time is from January 1, 0001, the greater the value of that point in time.

Comparisons and string representations involving time values always include seconds. If the string representation omits seconds, zero seconds is implied.

Comparisons involving timestamp values are chronological without regard to representations that could possibly be considered equivalent. The following predicate is therefore true:

```
TIMESTAMP('1985-02-23-00.00.00') > '1985-02-22-24.00.00'
```

## Literals

A literal specifies a value. Literals are classified as:

1. Character string literals (CHARACTER or CHAR), and
2. Numeric literals, which are further classified as:
  - **integer** (INTEGER or INT)
  - **decimal** (DECIMAL or DEC and NUMERIC)
  - **floating-point** (FLOAT, REAL and DOUBLE PRECISION)

## Character String Literals

A character string literal is a string that specifies a character string.

**Note:** String literals may include Double-Byte Character Set (DBCS) characters. The DBCS string must be preceded by the Shift-Out character and ended with the Shift-In character (the Shift characters are specified in the CXXMAINT option of the CA Datacom/DB Utility (DBUTLTY)).

There are two forms of character string literals.

## First Form

The first form is a sequence of characters that starts and ends with a string delimiter as specified by the STRDELIM= plan option. A string delimiter is either an apostrophe (') or a quotation mark (").

This form of string literal specifies the character string contained between the string delimiters. Two consecutive string delimiters are used to represent one string delimiter within the character string. The length of the character string must not be greater than 32720. Character string literals are delimiter tokens.

Some examples of character string literals are:

- 'YES'
- '32'
- 'DON''T CHANGE'

Dates, times, and timestamps are special character string literals with minimum length requirements as described in the following sections:

### Dates

Dates must have a minimum of 10 characters. Formats for dates are as follows (where yyyy represents the year, mm the month, and dd the day):

- ISO is yyyy-mm-dd (International Standards Organization).
- USA is mm/dd/yyyy (IBM USA Standard).
- EUR is dd.mm.yyyy (IBM European Standard).
- JIS is yyyy-mm-dd (Japanese Industrial Standard).

You can vary the formats for dates as given previous as follows:

- Trailing blanks are the only other characters allowed.
- Leading zeros may be omitted from the month and day.

### Times

Times must have a minimum of eight characters. Formats for times are as follows (where hh represents hours, mm minutes, and ss seconds):

- ISO is hh.mm.ss
- USA is hh:mm AM or PM
- EUR is hh.mm.ss
- JIS is hh:mm:ss

You can vary the formats for times as given previous as follows:

- You can omit leading zeros from the hour.
- Omission of seconds implies zero seconds.
- You may omit minutes from the USA format. Omission of minutes implies zero minutes.
- Trailing blanks are the only other characters allowed.

#### **Timestamps**

Timestamps must have a minimum of 26 characters. The format for timestamps in all cases is as follows (where yyyy-mm-dd represents the date, hh.mm.ss the time, and nnnnnn the microseconds):

- yyyy-mm-dd-hh.mm.ss.nnnnnn

You can vary the formats for times as given previous as follows:

- You may omit leading zeros from the month, day, and hour.
- Trailing blanks are the only other characters allowed.

## Second Form (HEX)

The second form of character string literals begins with an X, followed by a sequence of characters that starts and ends with a string delimiter. The characters between the string delimiters must be an even number of hexadecimal digits. A hexadecimal digit is either a digit or any of the letters: A B C D E or F (upper or lower case).

This form of string literal allows you to specify characters that do not have a keyboard representation (for example, X'FFFF' or X'01DABC').

## Integer Literals

An integer literal specifies a binary integer which can be signed, is no greater than 10 digits and does not include a decimal point.

The data type of an integer literal is INTEGER (large integer). The value must be within the range of a large integer. Examples of integer literals are:

- 32
- -10
- +255



## Floating Point Literals

A floating-point literal specifies a floating-point number as two numbers separated by an E. The first number may include a decimal point and a sign. The second number may include a sign, but not a decimal point.

The data type of a floating-point literal can be `FLOAT`, `REAL` or `DOUBLE PRECISION`. The value of the literal is the product of the first number and the power of 10 specified by the second number. The value must be within the range of floating-point numbers.

The number of characters in the literal must not exceed 30. Excluding leading zeros:

- The number of digits in the first number must not exceed 17.
- The number of digits in the second number must not exceed 2.

Examples of floating-point literals are:

- 10E1
- 3.E4
- 3.2E-2
- 7.E+3

## Decimal Literals

A decimal literal specifies a `DECIMAL` or `NUMERIC` data type number which can be signed, can be no greater than 31 digits and includes a decimal point.

The precision is the total number of digits, including leading and trailing zeros. The scale is the number of digits to the right of the decimal point, including trailing zeros.

Examples of decimal literals are:

- 1.5
- 1000.
- -100.
- +144.

## Column Names

A column-name designates a column of a table or view. The name can be unqualified or qualified.

If you specify `SQLMODE=ANSI` or `SQLMODE=FIPS` in the SQL Preprocessor options, the unqualified form of a column name is a long identifier, 1 to 18 characters.

If you specify `SQLMODE=DATACOM` for extended mode in the SQL Preprocessor options, the unqualified form of a column name can be 1 to 32 characters.

The column-name must be alphanumeric and the first character must be alphabetic. The name can contain underscores ( `_` ), but not hyphens ( `-` ).

Do not use a keyword as the name of a column.

The meaning of a column-name depends on its context. A column-name can be used to:

- Declare the name of a column, as in the `CREATE TABLE` statement.
- Identify a column, as in the `SELECT` statement.
- Specify values of a column, as in the following contexts:

### **In functions**

A column-name specifies all values of the column in the group or intermediate result table to which the function is applied.

For example, `MAX(SALARY)` applies the function `MAX` to all values of the column `SALARY` in a group.

### **In GROUP BY or ORDER BY clauses**

A column-name specifies all values in the intermediate result table to which the clause is applied.

For example, `ORDER BY DEPT` orders an intermediate result table by the values of the column `DEPT`.

### **In expressions or search conditions**

A column-name specifies a value for each row or group to which the expression or search condition is applied.

For example, when the search condition `CODE = 20` is applied to a row, the value specified by the column-name `CODE` is the value of the column `CODE` in that row.

## Qualified Column Names

A qualifier for a column-name can be a:

- Table-name
- View-name
- Synonym
- Correlation-name

When you qualify a column name, you enter the qualifier, followed immediately by a period (.), then the column-name. The format for qualifying a column name is shown in the below:

```
qualifier.column-name
```

Whether a column-name may be qualified depends on its context. In some forms of the COMMENT ON statement, a column-name must be qualified. Where the column-name specifies values of the column, it may be qualified at the user's option.

In all other contexts, a column-name must not be qualified.

Where a qualifier is optional, it can serve two purposes. For a discussion of these purposes, see [Column-Name Qualifiers to Avoid Ambiguity](#) (see page 517) and [Column-Name Qualifiers in Correlated References](#) (see page 518).

## Correlation Names

A correlation-name can be defined in any FROM clause.

For example, the clause

```
      .  
      .  
FROM X.MYTABLE Z, Y.MYTABLE  
      .  
      .
```

Establishes Z as a correlation-name for X.MYTABLE. Z can be used anywhere in the SELECT statement to designate that particular table.

A correlation-name is associated with a table or view only within the context in which it is defined. The same correlation-name can be defined for different purposes in different statements, or in different clauses of the same statement.

As a qualifier, a correlation-name can be used:

- To avoid ambiguity
- To establish a correlated reference
- As a shorter name for a table or view

In the previous example, Z could be used to avoid having to enter X.MYTABLE more than once.

## SQL Index Binding

Beginning in r11, SQL binds to a specific key more frequently than in previous versions. The purpose of this is to reduce the cost of key selection optimization in Compound Boolean Selection at execution.

You can tell if a key was selected during binding by the SQL Optimizer in the Compound Boolean Selection (CBS) Diagnostic Report when the KEY parameter specifies a key name. This is also reflected in the CBSOR Accounting Element.

You may directly specify the key to use in your query by appending "\_HINT\_keynm" to the correlation name, where keynm is the 5-character internal key name that is printed in the SQL Optimization and CXX Reports.

**Note:** If the 5-character key name is not found or cannot be used due to KEY\_INC = N (that is, nil values not indexed), the query is processed as if no HINT key had been given.

In the following example, assume that:

- CARS is the table name.
- COLOR is the 5-character key name (keynm).
- T1 is the variable chosen to prefix \_HINT\_keynm, that is, in this specific example T1 is the prefix of the SQL identifier that composes the correlation name, resulting in T1\_HINT\_COLOR.
- Indexes exist for COLOR and MAKE.

If there are more MAKE values than COLOR values, the SQL Optimizer may select the MAKE index, but in the case of this example, because the COLOR is so rare, it is the best key to use.

Following is the coding specific to this example, given the previous explanation.

```
SELECT *
FROM CARS T1_HINT_COLOR
WHERE T1_HINT_COLOR.COLOR = "PINK"
AND T1_HINT_COLOR.MAKE = "FORD";
```

## Column-Name Qualifiers to Avoid Ambiguity

In the context of a function, GROUP BY clause, ORDER BY clause, expression or a search condition, a column-name refers to values of a column in some table or view.

The tables and views that could possibly contain the column are called the object tables of the context. Two or more object tables could possibly contain columns with the same name.

One reason for qualifying a column-name is to designate which table the column comes from.

## Table Designators

A qualifier that designates a specific object table is a table designator. The clause that identifies the object tables also establishes the table designators for these tables.

For example, the object tables of an expression in a SELECT clause are named in the FROM clause that follows the SELECT, as in this partial statement:

```
SELECT Z.CODE, MYTABLE.CODE

FROM X.MYTABLE Z, MYTABLE,

WHERE . . .
```

A name that follows a table or view name is a correlation-name and a table designator. In the previous example, Z is a table designator and qualifies the first column name after SELECT.

A table-name, view-name or synonym that is not followed by a correlation-name is a table designator. In the previous example, MYTABLE is a designator and qualifies the second column-name after SELECT.

## Avoiding Undefined or Ambiguous References

When a column-name refers to values of a column, exactly one object table must include a column with that name.

The following conditions are considered errors:

- No object table contains a column with the specified name. In this case, the reference is undefined.
- The column name is qualified by a table designator, but the designated table does not include a column with the specified name. Again, the reference is undefined.
- The name is unqualified and more than one object table includes a column with that name. In this case, the reference is ambiguous.

Avoid ambiguous references by qualifying a column-name with a uniquely defined table designator.

If the column is contained in several object tables with different names, the table names can be used as designators.

In the case that two or more of the object tables are instances of the same table and have the same name, you can use correlation-names to designate, unambiguously, the particular instances of the table.

For example, in the following FROM clause, X and Y are defined to refer, respectively, to the first and second instances of the table NAMETBL.

```
      .  
      .  
FROM NAMETBL X, NAMETBL Y  
      .  
      .
```

## Column-Name Qualifiers in Correlated References

A subselect used with a search condition is a subquery. A subquery is said to be at a lower level than (or nested in) the subselect that contains the search condition.

A subquery can contain search conditions of its own. Among the object tables of those search conditions are all the tables and views identified at any higher level.

Some of the higher level tables and views may be other instances of the same tables or views named in the subquery. It may be necessary to refer specifically to a column in a table or view identified at a higher level. The means for doing so is a correlated reference.

A qualified column-name, QUAL.COLUMN, is a correlated reference if, and only if, these conditions are met:

- QUAL.COLUMN is used in a search condition of a subquery.
- QUAL.COLUMN is not used in the FROM clause of that subquery.
- QUAL is used in a FROM clause at some higher level.

QUAL.COLUMN refers specifically to the value of column COLUMN in table (or view) QUAL at the level where QUAL is used in the FROM clause. If QUAL is used in the FROM clause of more than one level, then QUAL.COLUMN refers to the level that most directly contains the subquery that contains QUAL.COLUMN.

The correlated reference QUAL.COLUMN identifies a value of COLUMN in a row or group of QUAL to which two search conditions are being applied. The conditions are located as follows:

1. Condition 1 is in the subquery.
2. Condition 2 is at some higher level.

If condition 2 is used in a WHERE clause, the subquery is evaluated for each row to which condition 2 is applied.

If condition 2 is used in a HAVING clause, the subquery is evaluated for each group to which condition 2 is applied.

For example:

```
SELECT EMPNO, LASTNAME, WORKDEPT
FROM NAMETBL X
WHERE SALARY > (SELECT AVG(SALARY)
                FROM NAMETBL
                WHERE WORKDEPT = X.WORKDEPT)
```

The previous statement lists employees who make more than the average salary for their department. The first FROM clause establishes X as a correlation-name for NAMETBL. The correlated reference X.WORKDEPT (in the last line) refers to the value of WORKDEPT in table NAMETBL at the level of the first FROM clause.

## Host Variables

A host variable is a data item that is referenced in an SQL statement. For information about how to define host variables in the source code:

- For COBOL, see [Host-Variable Definitions in COBOL](#) (see page 178).
- For PL/I, see [Host-Variable Declarations for PL/I](#) (see page 187).
- For Assembler, see [Host Variable Declarations for Assembler](#) (see page 201).
- For the C language, see [Rules for Coding Host-Variables in C](#) (see page 206).

The term host-variable, as used in the syntax diagrams, shows a reference to a host variable. A host variable in the INTO clause of a FETCH or SELECT statement identifies a host variable to which a value from a row is assigned.

In all other contexts, a host variable specifies a value.

The general form of a host variable reference is:

`:host-identifier`

The host-variable name must:

1. Be a valid data name in the host language that is being used.
2. Be defined as a legal elementary item with an acceptable embedded SQL data type.
3. Not include any spaces.

An indicator variable must be associated with a host variable if the host variable will become NULL. See [Indicator Variables](#) (see page 524) for more information.

## Host Structures

Host structures are allowed in COBOL and PL/I but not in Assembler.

You can qualify a host variable by a group name, for example:

`:GROUPNAME.VARNAME`

In this example, GROUPNAME is the host structure name and is used to qualify a host variable name. Host structure names may also be used in certain contexts to represent a list of the elementary items that they contain.



A host structure is a group whose subordinate levels are elementary data items. Host structures have a maximum of two levels, even though the structure could possibly itself occur within a multilevel structure. An exception is that the declaration of a varying-length character string variable requires another level, which must be level-49. In COBOL, for example:

```
01 GROUPX.
  02 SUBGROUP.
    03 C1  PIC X(4) .
    03 C2  PIC X(5) .
    03 C3.
      49 C3LENGTH PIC S9(4) COMP.
      49 C3STRING PIC X(30) .
    03 C4  PIC X(10) .
```

In the previous example, SUBGROUP is a valid host structure, because there is only one level subordinate to it, except for the varying-length variable. GROUPX is not a valid host structure, because it has more than one level beneath it.

Specifying a host structure is a short-hand for specifying a list of the elementary items it contains. For example:

```
EXEC SQL
SELECT COL1, COL2, COL3, COL4
  INTO :SUBGROUP
  FROM TABLEX
END-EXEC
```

Is equivalent to:

```
EXEC SQL
SELECT COL1, COL2, COL3, COL4
  INTO :SUBGROUP.C1, :SUBGRO.C2, :SUBGROUP.C3, :SUBGROUP.C4
  FROM TABLEX
END-EXEC
```

The Preprocessor does this expansion whenever a host variable is specified which qualifies as a valid host structure.

The form of a host structure reference is identical to the form of a host variable reference. The reference :S1:S2 is a host structure reference if S1 designates a host structure. If S1 designates a host structure, S2 must be defined as a vector of small integer variables, as for example in COBOL:

```
03 S2  PIC S9(4) COMP OCCURS 6 TIMES.
```

As just shown, S1 is the main structure and S2 is its indicator structure.

A host structure may be referenced in any context where a list of host variables may be referenced. A host structure reference is equivalent to a reference to each of the host variables contained within the structure in the order in which they are defined in the host language structure declaration. The nth variable of the indicator structure is the indicator variable for the nth variable of the main structure.

If the main structure has x more variables than the indicator structure, the last x variables of the main structure do not have indicator variables. If the main structure has x less variables than the indicator structure, the last x variables of the indicator structure are ignored. These rules also apply if a reference to a host structure includes an indicator variable (not a structure) or if a reference to a host variable (not a structure) includes an indicator structure. If an indicator structure or variable is not specified, no variable of the main structure has an indicator variable.

## Extended Format for Host Variables in COBOL

The extended format for the host variable supports the COBOL construct of qualified names. However, use of the extended format differs from the COBOL construct. For example, if you define:

```
01 DATE1.  
   03 MONTH1 PIC 99  
   03 DAY1   PIC 99  
   03 YEAR1  PIC 99
```

you reference DAY1 in the COBOL construct as follows:

```
MOVE DAY1 OF DATE1 TO TEMP-DAY
```

In an SQL statement, you reference DAY1 as follows:

```
SELECT NEWDAY  
INTO :DATE1.DAY1
```

### *Host Variable Data Types*

See [Character Strings](#) (see page 495) for more information on host variable data types.

For more information about SQL data types, see [Data Types](#) (see page 485).

### Using Colons in Host Variables

Host variables must be preceded by a colon (:) if the host identifier is identical to an SQL reserved word. Colons are always required on indicator variables. When used, the colon can be preceded by a blank, an open parenthesis or a comma, as shown in the following examples where **b** represents a blank.

```
b:host-variable  
(:host-variable  
,:host-variable
```

Colons may be omitted only in the following contexts:

- In the INTO clause of a:
  - SELECT INTO statement
  - FETCH statement
- Following LIKE or NOT LIKE
- Following VALUES (in an INSERT statement)
- Following IN (in a SELECT statement), when the target of the IN clause is a list of host variables and/or constants and/or special registers, NOT when the target of the IN is a subselect or an expression
- When an indicator variable is included (in the form '&V1V2.', NOT in the form ':V1 INDICATOR :V2')
- When qualified by a valid host structure name

**Note:** All except the last of the previous are those in which only host variables are allowed, not both column references and host variables. When both column references and host variable references are allowed, the colon should be used on host variables to avoid confusion, because CA Datacom/DB may not be able to give a clear indication of error when the colon is omitted. In a context in which either a host variable or column can be referenced, the use of an unqualified name without a colon is interpreted by the Preprocessor as a reference to a column, even if there is a host variable with that same name. If the host variable was what was intended, the results are confusing but no error is issued.

If a qualified name without a colon such as GROUPX.V is used, and GROUPX is a host structure that contains V, GROUPX.V is always interpreted by CA Datacom/DB as a reference to a host variable. If the group name happened to be the same as a table name and the host variable within the group was the same as a column name, that column could never be referenced because CA Datacom/DB would always interpret the name as the host structure reference. You should therefore avoid naming host structures the same as any possible qualifiers of a column-name that you could possibly specify in your program.

## Indicator Variables

Indicator variables associated with host variables are used to indicate the presence or absence of NULL values in the data. To avoid having to designate one of the valid values for each data type to represent the NULL value, for example, zero or blanks, a separate indicator variable is associated with each host variable. The indicator variable must be checked first. If the indicator variable contains -1, the value of the associated host variable is NULL. In this case, the value present in the host variable itself is undefined. If the indicator variable contains anything other than -1, the value of the associated host variable is not NULL; it is the value in the host variable itself.

An indicator variable must be associated with a host variable if the host variable will become NULL. An error message is issued if an indicator variable is not associated with a host variable and SQL attempts to assign a NULL value to the host variable. It is recommended that you always use indicator variables to avoid the possibility of encountering that error.

Provision is made in the CA Datacom/DB SQL Preprocessor for recognizing indicator variables.

An indicator variable must be defined as a SMALLINT data type. The indicator variable can be defined anywhere a host variable can be defined.

An indicator variable is associated with a host variable by the form:

`:host-variable:indicator-variable`

In the previous format:

- The first colon identifies a host variable.
- The second colon, appearing without a break, identifies an indicator variable associated with the host variable.

The rules for forming host variable names in an SQL statement apply equally to the host-variable and the indicator-variable. Both the host variable name and the indicator variable name can be qualified names. For more information on forming host variable names, see [Host Variables](#) (see page 520).

## SQL Parameters

An SQL parameter is a parameter that is passed to an SQL Procedure (a LANGUAGE SQL procedure). In SQL Procedures, SQL parameters can be used anywhere expressions are allowed. They cannot be used outside of procedures except as keyword parameters in the CALL and EXECUTE statements. When used in an SQL Procedure containing an SQL variable with a conflicting (matching) name, or in a statement containing a table or view reference where the table or view contains a conflicting column name, the name should be qualified using the procedure name. The syntax of an SQL parameter follows:

►► [ proc-name. ] SQL-parameter-name ◄◄

**Note:** The proc-name is discussed further in the CREATE PROCEDURE section. The proc-name can be qualified by an authorization ID.

The SQL-parameter-name is discussed further in the CREATE PROCEDURE section.

## SQL Variables

An SQL variable is a variable that is declared within a compound statement in an SQL Procedure (a LANGUAGE SQL procedure). In compound statements, SQL variables can be used anywhere expressions are allowed. They cannot be used outside of compound statements. If the name conflicts with (matches) another SQL variable name (for example from another compound statement such as a condition handler), an SQL parameter name, or a column contained within a referenced table or view, the SQL variable name should be qualified using the start-label of the compound statement that immediately contains it. The syntax of an SQL variable follows:

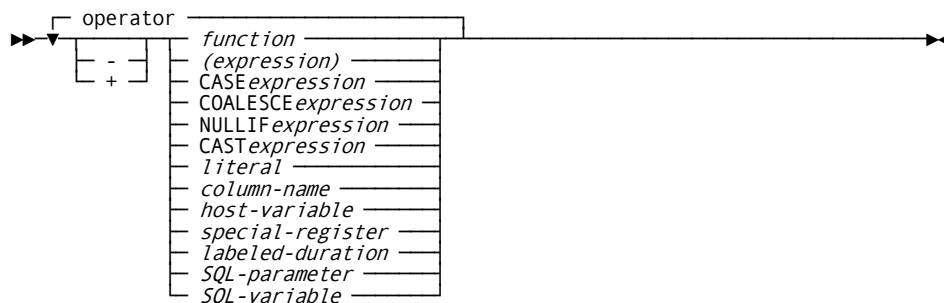
►► [ start-label. ] variable-name ◄◄



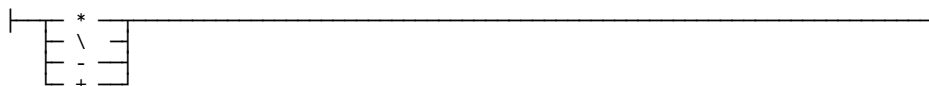
# Chapter 24: Expressions

---

An expression specifies a value. Following is the syntax diagram for an expression.



Expansion of Where operator is as follows



The special-register and labeled-duration are CA Datacom/DB extensions. See [Special Registers](#) (see page 533) and [Labeled Duration](#) (see page 535).

## **function**

Specify a function. For more information about functions, see [Functions](#) (see page 549).

## **(expression)**

Specify an expression.

## **CASE expression**

For information about CASE expressions, see [CASE Expressions](#) (see page 529).

## **COALESCE expression**

For information about COALESCE expressions, see [COALESCE and NULLIF Expressions](#) (see page 529).

## **NULLIF expression**

For information about NULLIF expressions, see [COALESCE and NULLIF Expressions](#) (see page 529).

## **CAST expression**

For information about CAST expressions, see [CAST Expressions](#) (see page 529).

***literal***

Specify a literal. If the expression is numeric, the literal must be numeric. For more information on literals, see [Literals](#). (see page 510)

***column-name***

Specify the name of a column in a table or view. If the expression is an arithmetic expression, the column must be of a numeric data type.

***host-variable***

Specify a host-variable. A host-variable in an expression must identify a variable described in the program under the rules for declaring host-variables. For more information on host-variables, see [Host Variables](#) (see page 520).

***special-register***

A special register can be used wherever an expression can be used, except that special registers may not be used in the search condition of a CHECK constraint. See [Special Registers](#) (see page 533).

***labeled-duration***

A labeled duration can only be used in an expression that involves a date or time value. See [Labeled Duration](#) (see page 535).

***SQL-parameter***

Inside SQL Procedures, an *SQL-parameter* can be used anywhere expressions are allowed. Except in the CALL and EXECUTE statements, an *SQL-parameter* cannot be used outside of procedures. For more information, see [SQL Parameters](#) (see page 525).

***SQL-variable***

Inside compound statements, an *SQL-variable* can be used anywhere expressions are allowed, but an *SQL-variable* cannot be used outside of a compound statement. For more information, see [SQL Variables](#) (see page 525).

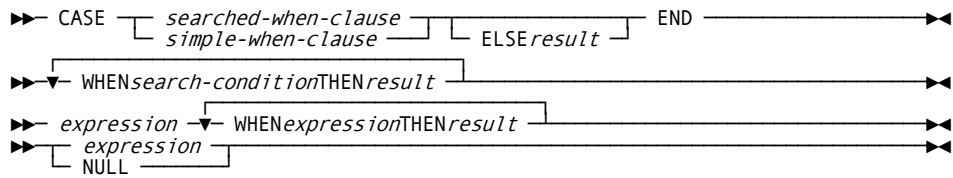


## CASE, COALESCE, NULLIF, and CAST

### CASE Expressions

Beginning in r11, CA Datacom/DB supports CASE expressions. The value of a CASE expression is the *result* of the first CASE that evaluates to TRUE. If no CASE evaluates to TRUE, the value is the *result* of the ELSE. If no ELSE is coded, the value is the NULL value.

**Note:** If a CASE evaluates to UNKNOWN because of NULL values, the effect is the same as if the CASE evaluated to FALSE, and the next CASE or ELSE is evaluated.



### CASE

Begins a CASE expression.

### *searched-when-clause*

Specifies a *search-condition* and the result when that condition is true.

### *simple-when-clause*

The value of the first *expression* is compared to each *WHEN expression*, and the CASE result is the first *WHEN* result that is TRUE. The data type of the *expressions* must be compatible.

### *result*

Specifies an expression that follows the THEN and ELSE keywords. There must be at least one *result* in the CASE expression that is not NULL. All *result* expressions must be compatible.

### *search-condition*

Specifies a condition that is true, false, or unknown about a row. The search condition cannot contain a subquery.

### END

Ends a CASE expression.

**Example 1 (Simple WHEN Clause)**

Using simple WHEN clauses, this example translates the response codes to a survey:

```
SELECT EMPNO, QUESTION,
       CASE RESPONSE_CODE
       WHEN 1 THEN 'Strongly Disagree'
       WHEN 2 THEN 'Somewhat Disagree'
       WHEN 3 THEN 'Neutral'
       WHEN 4 THEN 'Somewhat Agree'
       WHEN 5 THEN 'Strongly Agree'
       ELSE      'No Response'
       END
FROM SURVEY;
```

**Example 2 (Searched WHEN Clause)**

Using searched WHEN clauses, this example translates the response codes to a survey:

```
SELECT EMPNO, QUESTION,
       CASE
       WHEN MODE = YN AND RESPONSE_CODE = 1 THEN 'YES'
       WHEN MODE = YN AND RESPONSE_CODE = 2 THEN 'NO '
       ELSE 'NO RESPONSE'
       END
FROM SURVEY;
```

**COALESCE and NULLIF Expressions**

Beginning in r11, CA Datacom/DB supports COALESCE and NULLIF expressions, shortcuts for expressing two frequent uses of the CASE expression.

COALESCE is the ANSI 1999-compatible version of the VALUE scalar function. It returns the first non-NULL argument it is passed.

►► COALESCE – (  $\overbrace{\text{result-expression}}^{\text{result-expression}}$  ) ◄◄

NULLIF returns NULL if the arguments it is passed are equivalent. Otherwise, it returns the first argument.

►► NULLIF – ( *result-expression-1, result-expression-2* ) ◄◄

Coding:

**COALESCE(value1, value2)**

is the same as coding:

**CASE WHEN value1 IS NOT NULL THEN value1 ELSE value2 END**

**Example**

Using COALESCE, select the value of *parm*, which can be specified at the global, region or individual level:

```
SELECT COALESCE(I.PARM, R.PARM, G.PARM)
FROM GLOBAL G, REGION R, INDIVIDUAL I
WHERE I.REGION = R.REGION
      AND I.ID_NBR = :HST_ID_NBR
```

Specifying:

**NULLIF(value1, value2)**

is the same as specifying:

**CASE WHEN value1 = value2 THEN NULL ELSE value1 END****Example**

Using NULLIF, substitute NULLs for zeros as follows:

```
SELECT EMPNO, NULLIF(COMMISSION, 0) FROM EMPLOYEE
```

All arguments passed to COALESCE and NULLIF must be of compatible data types.

**CAST Expressions**

Beginning in r11, CA Datacom/DB supports CAST expressions to give users access to data stored in non-standard formats. Although CAST has a wide range of potential uses, users of CA Datacom Datadictionary REDEFINES should be particularly interested in the feature, because it enables SQL to access redefined data using the correct data type.

Consider the following example. Assume that a table named "customers" contains legacy data and has a column called "credit\_info" that SQL knows as a CHAR[7]. CA Datacom Datadictionary redefines the column as a CHAR[1] containing a credit "grade" of 'A' through 'F', followed by a 6-digit zoned decimal representing a credit limit in whole dollars. The credit limit can now be accessed as a number using the following syntax:

```
►► CAST ( <-source-expression>AS <-target-type>WITHOUT CONVERSION) ◄◄
```

In the syntax, *source-expression* refers to the source value expression, and *target-type* refers to the target data type.

WITHOUT CONVERSION is a CA Datacom extension that means the source data is already formatted as the target data type and does not need to be converted. Following is an example specification:

```
CAST(SUBSTR(credit_info,2,6) AS NUMERIC(6,0) WITHOUT CONVERSION)
```

CAST WITHOUT CONVERSION allows any data type to be cast to any other data type. However, it is your responsibility to ensure that the source data matches the target data type and is not longer than the target. Leading zeroes and trailing blanks are inserted into targets as needed. For example, casting a CHAR(2) containing 1234 hexadecimal to INTEGER without conversion results in 00001234 hexadecimal, and casting an INTEGER containing F1F2F3F4 hexadecimal to CHAR(6) without conversion results in "1234 " (note the trailing spaces), or F1F2F3F44040 hexadecimal.

Following are some examples to help clarify how to specify the target type for CAST WITHOUT CONVERSION:

<b>If your source data is:</b>	<b>The target type should be:</b>
Binary 2 bytes	SMALLINT
Binary 4 bytes	INTEGER
31 zoned decimal digits	NUMERIC(31)
6 packed decimal digits	DECIMAL(6)

Following is an example of the use of CAST in a query where the credit limit is retrieved as a number rather than a character string:

```
SELECT name, account_number,
       CAST(SUBSTR(credit_info,2,6) AS NUMERIC(6,0) WITHOUT CONVERSION)
       AS credit_limit
FROM customers
```

Following is an example showing a query that uses the credit limit in a mathematic comparison:

```
SELECT name, account_number
FROM customers
WHERE
CAST(SUBSTR(credit_info,2,6) AS NUMERIC(6,0) WITHOUT CONVERSION) > 50000
```

For the matrix (see [Numeric Assignments](#) (see page 502)) of implicit (automatic) data type conversions that CA Datacom/DB already supports, the following version of CAST may also be used:

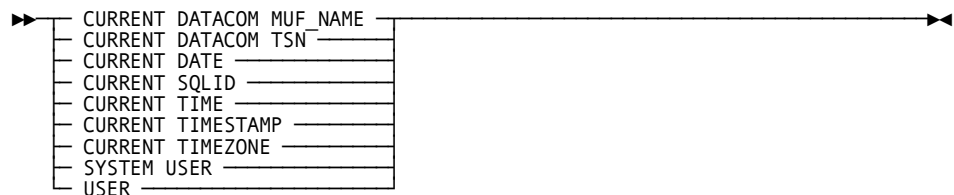
```
►► CAST ( (<- source-expression>AS <- target-type>) ◀◀
```

This version of CAST requires that the source value be interpreted first and then stored into the target in converted form. For example, when casting zoned decimal (NUMERIC) to packed decimal (DECIMAL), the source would have to contain a valid zoned decimal value.

## Special Registers

A special register can be used wherever an expression can be used, except that special registers may not be used in the search condition of a CHECK constraint.

Following is the special register syntax diagram.



### **CURRENT DATACOM MUF\_NAME**

Produces a CHAR(8) containing the name of the Multi-User Facility (MUF) that is processing the request. This name can be specified in the MUF Multi-User startup option, or it defaults to the job name of the MUF.

### **CURRENT DATACOM TSN**

Produces an INTEGER containing the current value of the Transaction Sequence Number (TSN) of the current unit of work. If no maintenance (INSERT/UPDATE/DELETE) has been performed in this unit of work, the value returned is zero.

### **CURRENT DATE**

Represents the current local date. It is derived from a reading of the time-of-day clock plus the CURRENT TIMEZONE.

### **CURRENT SQLID**

Represents the current SQL authorization ID, that is, the authorization ID (in this case actually a schema ID).

### **CURRENT TIME**

Represents the current local time. It is derived from a reading of the time-of-day clock plus the CURRENT TIMEZONE.

### **CURRENT TIMESTAMP**

Represents the current local timestamp. It is derived from a reading of the time-of-day clock plus the CURRENT TIMEZONE.

### **CURRENT TIMEZONE**

Is defined as DECIMAL(6,0) and represents the number of hours, minutes and seconds from Greenwich, England because Greenwich time is Greenwich Mean Time (GMT). East of Greenwich the CURRENT TIMEZONE number is negative. West of Greenwich the CURRENT TIMEZONE number is positive. The CURRENT TIME minus the CURRENT TIMEZONE gives the GMT.

### **SYSTEM USER**

Is the accessor ID of the currently signed on user.

The accessor ID is padded on the right with blanks, if necessary, so that the value of SYSTEM USER is always a fixed-length character string of length 18.

In the following example, the result table of the SELECT contains all rows where the value of the column CREATOR is equal to the accessor ID of the user who executes the query.

```
SELECT * FROM CA.INVENTIONS WHERE CREATOR = SYSTEM USER
```

**Important!** The CA Datacom/DB Security Facility must be installed at your site for CA Datacom/DB to determine the accessor IDs of users.

### **USER**

Is the current authorization ID. This is the same as the authorization ID of the currently executing plan.

## Labeled Duration

A labeled duration represents any number of years, months, days, hours, minutes, seconds, or microseconds. This number is then converted as if it were assigned to a `DECIMAL(15,0)`. The unit is expressed by a keyword following the number. For example, `25 YEARS` is the labeled duration in `HIREDATE + 25 YEARS`.

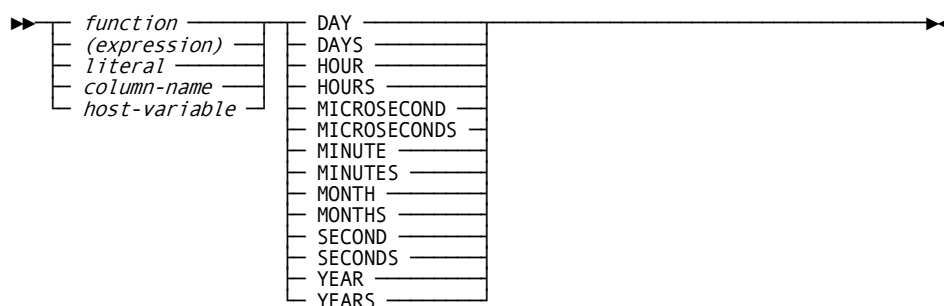
A labeled duration can only be used in an expression that involves a date or time value. For example:

`HIREDATE + 25 YEARS + 1 MONTH` is valid, but

`HIREDATE + (25 YEARS + 1 MONTH)` is not valid.

Following is the syntax diagram for labeled durations.

**Note:** Labeled duration, this is a CA Datacom/DB extension.



### ***function***

Specify a function. For more information about functions, see [Functions](#) (see page 549).

### ***(expression)***

Specify an expression. For more information about expressions, see [Expressions](#) (see page 527).

### ***literal***

Specify a literal. If the expression is numeric, the literal must be numeric. For more information on literals, see [Literals](#). (see page 510)

***column-name***

Specify the name of a column in a table or view. If the expression is an arithmetic expression, the column must be of a numeric data type.

***host-variable***

Specify a host-variable. A host-variable in an expression must identify a variable described in the program under the rules for declaring host-variables. For more information on host-variables, see [Host Variables](#) (see page 520).

**DAY/DAYS**

A duration expressed in day(s). For more information on durations, see [Durations](#) (see page 542).

**HOUR/HOURS**

A duration expressed in hour(s). For more information on durations, see [Durations](#) (see page 542).

**MICROSECOND/MICROSECONDS**

A duration expressed in microsecond(s). For more information on durations, see [Durations](#) (see page 542).

**MINUTE/MINUTES**

A duration expressed in minute(s). For more information on durations, see [Durations](#) (see page 542).

**MONTH/MONTHS**

A duration expressed in month(s). For more information on durations, see [Durations](#) (see page 542).

**SECOND/SECONDS**

A duration expressed in second(s). For more information on durations, see [Durations](#) (see page 542).

**YEAR/YEARS**

A duration expressed in year(s). For more information on durations, see [Durations](#) (see page 542).



## Expressions without Arithmetic Operators

If the arithmetic operators are not used, the result of the expression is the specified value. Some examples of expressions without arithmetic operators are:

- `COST` -- column-name
- `:COST` -- host variable
- `'COST'` -- string literal
- `MIN(COST)` -- function performed with column-name as argument

## Expressions with the Concatenation Operator

The concatenation operator is represented by two vertical lines (`||`) or by `CONCAT`. If the concatenation operator is used in an expression, the result of the expression is a character string. The operands of concatenation must be character strings.

If either operand can be null, the result of the expression can be null. If either operand is null, the result of the expression is the null value. The result otherwise consists of the first operand character string followed by the second operand character string. The length of the result character string is the sum of the lengths of the operand strings.

In a concatenation, if any operand is `MIXED`, the result of the expression is `MIXED`. When concatenating a `MIXED` string ending with Shift-In with a `MIXED` string beginning with Shift-Out, both Shift characters are omitted in the result.

Following is an example of using a concatenation operator:

```
LASTNAME || ', ' || FIRSTNAME
```

## Expressions with Arithmetic Operators

If arithmetic operators are used, the result of an expression is derived by the application of the operators to the value of the operands.

Arithmetic operators must not be applied to character strings. Thus, `USER + 2` is invalid.

The two types of arithmetic operators are prefix operators and infix operators.

### Unary Operators

The prefix operator `+` (unary plus) does not change its operand.

The prefix operator `-` (unary minus) reverses the sign of its operand. For example, if the data type of `A` is small integer, then the data type of `-A` is large integer.

The first character of the token following a prefix operator must not be a plus or minus sign. For example, `--10` is not allowed since `10` is preceded by two minus signs.

**Note:** Unary operators are not allowed with `FLOAT`, `REAL` and `DOUBLE PRECISION` data types.

### Infix Operators

The infix operators and the operation they specify are:

Operator	Operation
<code>+</code>	Addition
<code>-</code>	Subtraction
<code>*</code>	Multiplication
<code>/</code>	Division

The operand of an infix operator must not be a function that includes the keyword `DISTINCT`.

The value of the second operand of division must not be zero.

### Conversions During Arithmetic Operations

Conversion implies that each of the two values are represented in a form compatible for the arithmetic operation.

The following table lists the conversions that must take place before any arithmetic operation can occur.

To determine the data type used in the arithmetic operation, locate the data type of the first operand in the left column. Next, locate the data type of the second operand in the vertical columns to the right.

The data type specified at the intersection of the two columns is the data type used in the actual arithmetic operation. In some cases, this data type is different from that of both operands, indicating that both operands are converted before the arithmetic operation is performed.

Conversions For Arithmetic Operations														
	Second Operand:													
First Operand:	F	R	DP	D	SI	LI	N	V	C	DT	T	TS	G	VG
F	X	F	F	F	F	F	F	●	●	●	●	●	●	●
R	F	X	F	F	F	F	F	●	●	●	●	●	●	●
DP	F	F	X	F	F	F	F	●	●	●	●	●	●	●
D	F	F	F	X	D	D	D	●	●	●	●	●	●	●
SI	F	F	F	D	X	LI	LI	●	●	●	●	●	●	●
LI	F	F	F	D	LI	X	D	●	●	●	●	●	●	●
N	F	F	F	D	D	D	D	●	●	●	●	●	●	●
V	●	●	●	●	●	●	●	X	X	●	●	●	●	●
C	●	●	●	●	●	●	●	X	X	●	●	●	●	●
DT	●	●	●	●	●	●	●	●	●	X	●	●	●	●
T	●	●	●	●	●	●	●	●	●	●	X	●	●	●
TS	●	●	●	●	●	●	●	●	●	●	●	X	●	●
G	●	●	●	●	●	●	●	●	●	●	●	●	X	X
VG	●	●	●	●	●	●	●	●	●	●	●	●	●	X

*Two Integer Operands*

If both operands of an arithmetic operator are integers, the operation is performed in binary and the result is INTEGER (large integer). Any remainder of division is lost.

The result of a binary arithmetic operation must be within the range of large integers.

*Integer and Decimal Operands*

If one operand is an integer and the other is DECIMAL (packed decimal) or NUMERIC (zoned decimal), the operation is performed in DECIMAL (packed decimal).

The operation uses a temporary copy of the integer which has been converted to a DECIMAL number with scale 0. The precision of the temporary copy depends on the characteristics of the operand as shown in the following:

<b>Operand</b>	<b>Precision of Decimal Copy</b>
Column or variable: large integer	11
Column or variable: small integer	5
Literal or column more than 5 digits (including leading zeros)	same
Literal: 5 digits or fewer	5

*Two Decimal Operands*

If both operands are DECIMAL (packed decimal) or NUMERIC (zoned decimal), the operation is performed in DECIMAL (packed decimal). The result of any decimal arithmetic operation is a decimal number with a precision and scale that are dependent on the operation and the precision and scale of the operands.

If the operation is addition or subtraction, and the operands do not have the same scale, the operation is performed with a temporary copy of one of the operands. The temporary copy is extended with trailing zeros so that its fractional part has the same number of digits as the other operand.

The result of a decimal operation must not have a precision greater than 31.

The result of decimal addition, subtraction and multiplication is derived from a temporary result, which may have a precision greater than 31.

- If the precision of the temporary result is not greater than 31, the final result is the same as the temporary result.
- If the precision of the temporary result is greater than 31, the final result is derived from the temporary result by the elimination of leading digits so the final result has a precision of 31. The eliminated digits must all be zeros.

Following are formulas defining the precision and scale of the result of decimal operations in SQL.

The following table lists the symbols designating the precision and scale for each operand.

Operand	Precision	Scale
first operand	p	s
second operand	p'	s'

The following table lists the formulas for the precision and scale for the four arithmetic operations.

<b>Operation:</b>	<b>Addition and Subtraction</b>
<b>Precision:</b>	$\min(31, \max(p - s, p' - s') + \max(s, s') + 1)$
<b>Scale:</b>	$\max(s, s')$
<b>Operation:</b>	<b>Multiplication</b>
<b>Precision:</b>	$\min(31, p + p')$
<b>Scale:</b>	$\min(31, s + s')$
<b>Operation:</b>	<b>Division</b>
<b>Precision:</b>	31
<b>Scale:</b>	If $s' \leq 15$ then $\text{scale} = (m - p') - p - s + s'$ else $\text{scale} = \max(s' - p' + 15, 0) + 15 - (p - s)$ endif $\text{scale} = \max(\text{scale}, 3)$ $m = 29$ if p is even, or $m = 30$ if p is odd

### *Floating-point Operands*

If either operand of an arithmetic operator is floating-point, the operation is performed in floating-point. Thus, if any element of an expression is a floating-point number, the result of the expression is a floating-point number.

An operation involving a floating-point number and a integer is performed with a temporary copy of the integer. The temporary copy of the integer is converted to floating-point.

An operation involving a floating-point number and a DECIMAL (packed decimal) or NUMERIC (zoned decimal) number is performed with a temporary copy of the decimal number. The temporary copy of the decimal number is converted to floating-point.

The result of a floating-point operation must be within the range of floating-point numbers.

## Arithmetic Operations for Dates, Times, and Timestamps

Addition and Subtraction are the only arithmetic operations which you can perform on date and time values. You can increment or decrement a date, time or timestamp by a duration, or subtract a date from a date or a time from a time, but you cannot subtract a timestamp from a timestamp.

### Durations

A duration is a number that represents an interval of time. The number may be any of the following:

- Function
- Expression
- Literal
- Column name
- Host variable
- Special registers
- Labeled durations

Special registers and labeled durations are CA Datacom/DB extensions. See [Special Registers](#) (see page 533) for information on special registers. See [Labeled Duration](#) (see page 535) for information on labeled durations.

**Date Durations**

A date duration represents a number of years, months, and days. A date duration is expressed as DECIMAL(8,0). For example, a date duration of 1999 years, 3 months, and 20 days is 19990320.

Subtracting dates results in a date duration.

**Time Durations**

A time duration represents a number of hours, minutes and seconds. A time duration is expressed as DECIMAL(6,0). For example, a time duration of 19 hours, 30 minutes and 20 seconds is 193020.

Subtracting times results in a time duration.

**Addition Rules for Dates, Times, and Timestamps**

If operand one is a date, operand two must be a date duration or a labeled duration of years, months, or days.

If operand one is a time, operand two must be a time duration or a labeled duration of hours, minutes, or seconds.

If operand one is a timestamp, operand two must be a labeled duration, a date duration, or a time duration.

**Subtraction Rules for Dates, Times, and Timestamps**

The operation of subtracting two date-time values is different from the operation of subtracting a duration from a date-time value. The operands of date-time subtraction must be as follows:

- If operand one is a date, operand two must be a date, date duration, a date string, or a labeled duration of years, months, or days.
- If operand one is a time, operand two must be a time, time duration, a time string, or a labeled duration of hours, minutes, or seconds.
- If operand one is a timestamp, operand two must be a labeled duration, a time duration, or a date duration of years, months.
- If operand two is a date, operand one must be a date or date string.
- If operand two is a time, operand one must be a time or time string.
- A string representation of a date or time value cannot be subtracted from another string representation of a date or time value. However, if one of the strings is interpreted as a date or time, the expression is valid. For example:

`DATE('1989-03-10') - '1985-03-10'` is valid, but `('1989-03-10' - '1985-03-10')` is invalid.

### Using Durations to Increment or Decrement Dates

The result of adding or subtracting a duration to or from a date is a date. If *D* is a date and *N* is a number defined as `DECIMAL(15,0)`, the result of *D* + *N* YEARS or *D* - *N* YEARS is the date that is *N* years before or after *D*. Only years are counted. The month of the result is always the same as the month of *D*. The day of the result is the same as the day of *D*, unless the result is February 29th of a non-leap year, in which case the day part of the result is 28 and `SQLWARN6` is set to *W*.

The result of *D* + *N* MONTHS or *D* - *N* MONTHS is the date that is *N* months before or after *D*. Only months and years (if necessary) are counted. The day of the result is the same as the day of *D*, unless the result would be an invalid date, in which case the day part of the result is the last day of the month and `SQLWARN6` is set to *W*.

The result of *D* + *N* DAYS or *D* - *N* DAYS is the date that is *N* days before or after *D*.

If *N* is a duration of *y* years, *m* months, and *d* days, the result of *D* + *N* (where *N* is positive) or *D* - *N* (where *N* is negative) is the date that is *y* years, *m* months, and *d* days after *D*. The arithmetic is performed in this order using the previously defined rules, including the setting of `SQLWARN6` whenever an end-of-month adjustment is made.

If *N* is a duration of *y* years, *m* months, and *d* days, the result of *D* - *N* (where *N* is positive) or *D* + *N* (where *N* is negative) is the date that is *d* days, *m* months, and *y* years before *D*. The arithmetic is performed in that order using the previously defined rules, including the setting of `SQLWARN6` whenever an end-of-month adjustment is made.

### Subtracting Dates

If *D1* and *D2* are dates, the result of *D1* - *D2* is a date duration that gives the number of years, months, and days between the two dates. The data type of the result is `DECIMAL(8,0)`. If *D1* is greater than or equal to *D2*, *D2* is subtracted from *D1*. If *D1* is less than *D2*, *D1* is subtracted from *D2* and the sign of the result is negative. Given the example *D1* - *D2*, the rules for date subtraction are:

- If `DAY(D2)` is less than or equal to `DAY(D1)`, the day part of the result is equal to `DAY(D1) - DAY(D2)`.
- If `DAY(D2)` is greater than `DAY(D1)`, the day part of the result is equal to `N + DAY(D1) - DAY(D2)`, where *N* is the last day of `MONTH(D2)`. `MONTH(D2)` is incremented by 1.
- If `MONTH(D2)` is less than or equal to `MONTH(D1)`, the month part of the result is equal to `MONTH(D1) - MONTH(D2)`.
- If `MONTH(D2)` is greater than `MONTH(D1)`, the month part of the result is equal to `12 + MONTH(D1) - MONTH(D2)`. `YEAR(D2)` is incremented by 1.
- The year part of the result is equal to `YEAR(D1) - YEAR(D2)`.



### Special Considerations Relating to Date Arithmetic

Because of the differing number of days in each month of the year, adding a month to a given date does not always result in the same day of the next month. For example, adding one month to January 31 would yield February 31, which is not a valid date. The result is adjusted back to the last day of the month, February 28. Therefore,  $D + N \text{ MONTHS} - N \text{ MONTHS}$  is not always equal to  $D$ .

To avoid inconsistencies in date arithmetic caused by months, use days rather than months. For example,  $\text{DATE}(\text{DAYS}(D1) + \text{DAYS}(D2) - \text{DAYS}(D3))$  gives accurate results, but  $D1 + (D2 - D3)$  may not give accurate results.

### Incrementing and Decrementing Times by Durations

The result of adding or subtracting a duration to or from a time is a time. The result is in the range of times. If  $T$  is a time and  $N$  is a number defined as  $\text{DECIMAL}(15,0)$ :

- The result of  $T + N \text{ HOURS}$  or  $T - N \text{ HOURS}$  is a time that is  $N$  hours before or after  $T$ . Only hours are counted. The minute and second of the result are the same as the minute and second of  $T$ .
- The result of  $T + N \text{ MINUTES}$  or  $T - N \text{ MINUTES}$  is the time that is  $N$  minutes before or after  $T$ . Only minutes and hours (if necessary) are counted. The second of the result is the same as  $T$ .
- The result of  $T + N \text{ SECONDS}$  or  $T - N \text{ SECONDS}$  is the time that is  $N$  seconds before or after  $T$ .
- If  $N$  is a duration of  $h$  hours,  $m$  minutes, and  $s$  seconds, the result of  $T + N$  or  $T - N$  is the time that is  $h$  hours,  $m$  minutes, and  $s$  seconds before or after  $T$ . The arithmetic is performed using the previously defined rules.

### Subtracting Times

If  $T1$  and  $T2$  are times, the result of  $T1 - T2$  is a time duration that gives the number of hours, minutes, and seconds between the two times. The data type of the result is  $\text{DECIMAL}(6,0)$ . If  $T1$  is greater than or equal to  $T2$ ,  $T2$  is subtracted from  $T1$ . Otherwise,  $T1$  is subtracted from  $T2$  and the sign of the result is negative. Given the example of  $T1 - T2$  the rules for time subtraction are:

- If  $\text{SECOND}(T2)$  is less than or equal to  $\text{SECOND}(T1)$ , the seconds part of the result is equal to  $\text{SECOND}(T1) - \text{SECOND}(T2)$ .
- If  $\text{SECOND}(T2)$  is greater than  $\text{SECOND}(T1)$ , the seconds part of the result is equal to  $60 + \text{SECOND}(T1) - \text{SECOND}(T2)$ , and  $\text{MINUTE}(T2)$  is incremented by one.
- If  $\text{MINUTE}(T2)$  is less than or equal to  $\text{MINUTE}(T1)$ , the minutes part of the result is equal to  $\text{MINUTE}(T1) - \text{MINUTE}(T2)$ .
- If  $\text{MINUTE}(T2)$  is greater than  $\text{MINUTE}(T1)$ , the minutes part of the result is equal to  $60 + \text{MINUTE}(T1) - \text{MINUTE}(T2)$ , and  $\text{HOUR}(T2)$  is incremented by one.
- The hour part of the result is equal to  $\text{HOUR}(T1) - \text{HOUR}(T2)$ .

### Special Considerations Relating to Time Arithmetic

Adding 24 hours to the time 00.00.00 results in 24.00.00.

Adding 24 hours to any other time results in the same time.

For example, adding 24 hours to 00.00.59 results in 00.00.59.

### Incrementing and Decrementing Timestamps by Durations

The result of adding or subtracting to or from a timestamp is a timestamp.

The date part of the arithmetic is performed using the rules previously defined for incrementing or decrementing a date by a date duration.

The time part of the arithmetic is performed using the rules previously defined for incrementing or decrementing a time by a time duration, except that any overflow or underflow of hours is carried into the date part of the result.

The following rules apply to microseconds:

- If S is a timestamp and N is a number, the result of S + N MICROSECONDS or S - N MICROSECONDS is the timestamp that is N microseconds before or after S.
- If S is a timestamp and N is a date duration of y years, m months, and d days, the result of S + N or S - N is the timestamp that is y years, m months, and d days before or after S.
- If S is a timestamp and N is a time duration of h hours, m minutes, and s seconds, the result of S + N or S - N is the timestamp that is h hours, m minutes, and s seconds before or after S. The microsecond part of the result is the same as the microsecond part of S.

## Precedence of Operations

Expressions within parentheses are evaluated first. When the order of evaluation is not specified by parentheses, operations are done in the following order.

1. Prefix operators (unary + or unary -)
2. multiplication and division
3. Addition and subtraction

Operators at the same level are evaluated from left to right.

The examples that follow show the difference in results when you use parentheses to specify an order of evaluation in an expression. The same equation is used in each example, but parentheses are either omitted or inserted to show that different results are possible.

**Example 1: RESULT =  $10 + 20 / 5 - 1 * 2$** 

RESULT = $10 + 20 / 5 - 1 * 2$	Division is performed since it is the first operation with the highest order of precedence going left to right in the expression.
RESULT = $10 + 4 - 1 * 2$	Multiplication is performed next since it has the highest order of precedence among the remaining operations.
RESULT = $10 + 4 - 2$	Addition is performed since it is the first operation going from left to right among the remaining operations.
RESULT = $14 - 2$	Subtraction is performed since it is the remaining operation.
<b>RESULT = 12</b>	

**Example 2: RESULT =  $(10 + 20) / 5 - 1 * 2$** 

RESULT = $(10 + 20) / 5 - 1 * 2$	Addition is performed since it is enclosed in parentheses.
RESULT = $30 / 5 - 1 * 2$	Division is performed next since it is the first operation with the highest order of precedence going from left to right in the expression.
RESULT = $6 - 1 * 2$	Multiplication is performed next since it has the highest order of precedence among the remaining operations.
RESULT = $6 - 2$	Subtraction is performed since it is the remaining operation.
<b>RESULT = 4</b>	

**Example 3: RESULT =  $((10 + 20) / 5 - 1) * 2$** 

RESULT = $((10 + 20) / 5 - 1) * 2$	Addition is performed since it is enclosed in the inner pair of parentheses.
RESULT = $(30 / 5 - 1) * 2$	Division is performed since it has a higher order of precedence than subtraction, even though both are enclosed within parentheses.
RESULT = $(6 - 1) * 2$	Subtraction is performed since it is enclosed in parentheses.
RESULT = $5 * 2$	Multiplication is performed since it is the remaining operation.
<b>RESULT = 10</b>	

### Examples

Some examples of expressions are:

**Example 1:** This example finds the maximum value for each instance of a salary added to a year-to-date commission.

```
MAX(SALARY + YTDCOMM)
```

**Example 2:** This example finds the number of distinct status codes, eliminating any duplicates.

```
COUNT(DISTINCT STATUS)
```

**Example 3:** This expression calculates the discount by multiplying the cost by 5 percent.

```
DISCOUNT = COST * .05
```

**Example 4:** This expression calculates the total amount of all discounts by summing the results for each instance when the cost is multiplied by 5 percent.

```
TOTDISC = SUM(COST * .05)
```

# Chapter 25: Functions

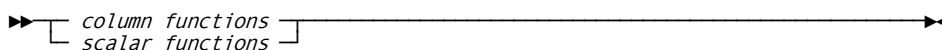
---

A function specifies a value. Functions are of two types, column functions and scalar functions. Scalar functions are a CA Datacom/DB extension.

Scalar functions can be nested within scalar functions or column functions, and column functions can be nested within scalar functions, but a column function cannot be nested within another column function. An expression in a column function must therefore not include another column function. See [Rules for Scalar Functions](#) (see page 555) for more information on nesting using scalar functions.

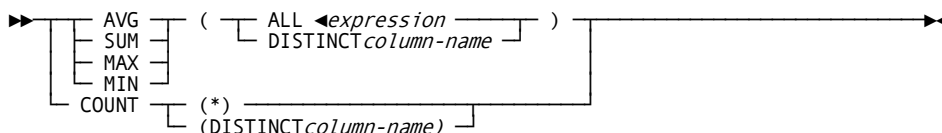
Following is the functions syntax diagram:

**Note:** Scalar functions are a CA Datacom/DB extension. See [Scalar Functions](#) (see page 554).



## Column Functions

Following is the syntax diagram for column functions:



## Description

### AVG

Specifies to return the average of the values in the column or expression. The values must be numeric.

### SUM

Specifies to return the sum of the values in the column or expression. The values must be numeric.

### MAX

Specifies to return the largest value in the column or expression.

**MIN**

Specifies to return the smallest value in the column or expression.

**ALL**

Specifies that duplicate values are not to be eliminated before the column function is applied. ALL is the default.

***expression***

Enter the expression which is to be the argument of the column function. An expression in a column function must include a column name (see the following rules). For more information about expressions, see [Expressions](#) (see page 527).

**DISTINCT**

Specifies that duplicate values are to be eliminated before the column function is applied.

***column-name***

Specifies the column which is the argument of the column function.

**COUNT(\*)**

Specifies to count all rows in the result table without eliminating duplicates.

When COUNT(\*) is the only item in the SELECT list and there is no join or WHERE clause, an optimization is used to take the count from the CXX rather than actually count the lines in the table. The potential exists for the CXX row count to be incorrect if the MUF has abended at a point where the latest CXX row count is only in memory, but this is an unlikely occurrence and the margin of error is small. In addition, actually counting the rows could potentially be off if inserts or deletes were issued after the count was begun. This enhancement provides a valuable and significant improvement in performance.

If the most accurate count is required, a column can be added to the SELECT list, or a WHERE clause could be added, for example, WHERE 1=1 would be sufficient.

**Example 1: CXX Count Returned**

```
SELECT COUNT(*) FROM SYSADM.AGGREGATE;
```

**Example 2: Count each row in table**

```
SELECT COUNT(*) FROM SYSADM.AGGREGATE WHERE 1 = 1;
```

**Note:** The CXX count could be inaccurate if MUF has abnormally terminated.

**COUNT(DISTINCT *column-name*)**

Specifies to return the number of distinct values of the named column.

## Rules for Column Functions

The result of a column function is derived by the application of the column function to the specified argument.

- The argument of `COUNT(*)` is a collection of rows.
- The argument of the other column functions is a collection of values. For `SUM` and `AVG`, the values must be numbers.
- The source of the argument of any column function is a group or an intermediate result table.

The following rules apply to all column functions other than `COUNT(*)`.

1. A column-name in a column function must not reference a column derived from a column function (a column of a view can be derived from a column function).
2. Column functions cannot be nested. Thus, an expression in a column function must not include a column function.
3. An expression in a column function must include a column-name. If the column-name is a correlated reference (which is allowed in a subquery of a `HAVING` clause), the expression must not include any operators.
4. Before a column function is applied, null values are eliminated from its argument.
  - If `DISTINCT` is specified, redundant duplicate values are also eliminated.
  - If `ALL` is specified, duplicates are not eliminated.
  - If neither `ALL` nor `DISTINCT` is specified, duplicates are not eliminated.

The following table lists each column function, the result of each, the data type of the result, and any exceptions.

**Column Functions:**

Column Function	Description
AVG	<p><b>Results:</b> The average of the values in its arguments. The values must be numbers. If the values are binary integers, the fractional part of the average is lost.</p> <p><b>Data Type:</b> The same as the data type of its argument.</p> <p><b>Exception:</b> The result is a large integer if the data type of the argument is a small integer. If the data type of the argument is decimal with precision <b>p</b> and scale <b>s</b>, the precision of the result is 31 and the scale is 31 - <b>p</b> + <b>s</b>. The sum of the values of the argument must be within the range of the result data type.</p>
COUNT(*)	<p><b>Results:</b> The number of rows in its argument.</p> <p><b>Data Type:</b> The result is always a large integer which cannot have a null value.</p>
COUNT(DISTINCT <i>column-name</i> )	<p><b>Results:</b> The number of values in its argument, that is to say, the number of distinct values of the column in the group or intermediate result table.</p> <p><b>Data Type:</b> The result is always a large integer.</p>
MAX	<p><b>Results:</b> The maximum value in its argument.</p> <p><b>Data Type:</b> The same as the data type of its argument.</p>
MIN	<p><b>Results:</b> The minimum value in its argument.</p> <p><b>Data Type:</b> The same as the data type of its argument.</p>



Column Function	Description
SUM	<p><b>Results:</b> The sum of the values in its arguments.</p> <p><b>Data Type:</b> The same as the data type of its argument. The values must be numbers.</p> <p><b>Exception:</b> The result is a large integer if the data type of the argument is a small integer. If the data type of the argument is decimal, the precision of the result is 31 and the scale is the same as the scale of the argument. The sum of the values must be within the range of the result data type.</p>

## Examples

The following examples show the use of column functions.

**Example 1:** This example shows how to find the average salary of all employees in the EMP table.

```
SELECT AVG(SALARY)
FROM EMP
```

**Example 2:** This example shows how to find the maximum and minimum salaries in the EMP table for employees whose age is over 30.

```
SELECT MAX(SALARY), MIN(SALARY)
FROM EMP
WHERE AGE > 30;
```

**Example 3:** This example finds the sum of salaries for all employees whose age is 30 or less.

```
SELECT SUM(SALARY)
FROM EMP
WHERE AGE <= 30;
```

**Example 4:** This example counts the number of shipments which included a part with the number P3. Duplicates are not eliminated.

```
SELECT COUNT(*)
FROM SHIPPART
WHERE PNUM = 'P3'
```

**Example 5:** This example counts the exact number of part numbers contained in PARTLIST. Any duplicates are eliminated.

```
SELECT COUNT(DISTINCT PNUM)
FROM PARTLIST
```

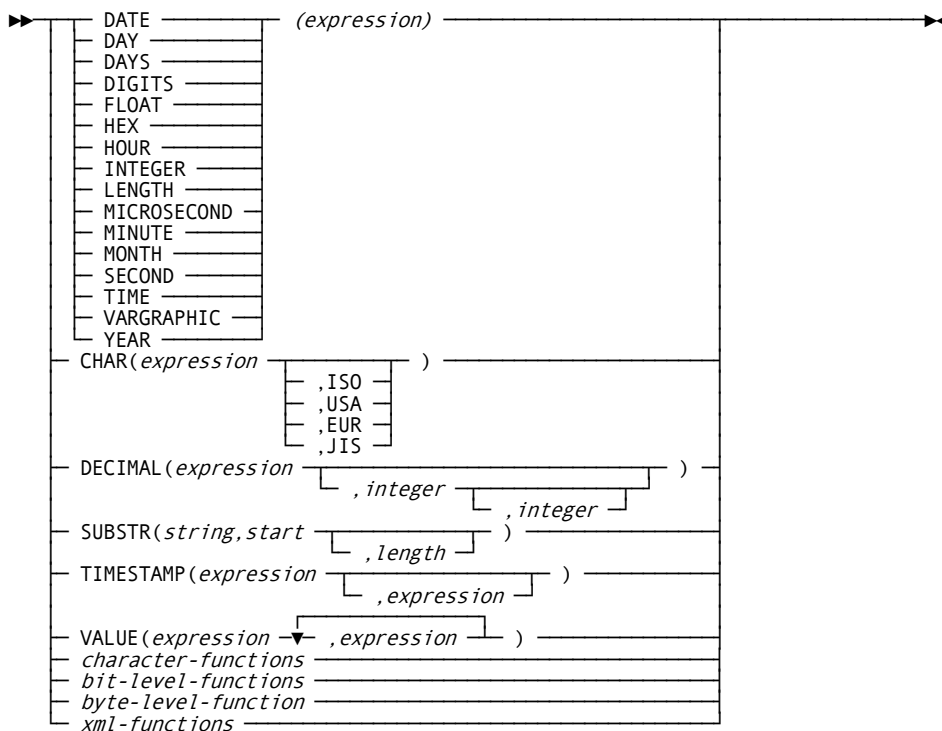
## Scalar Functions

Scalar functions are a CA Datacom/DB extension. A scalar function produces a single value from another value. It is expressed in the form of a function name, followed by a list of arguments enclosed in parentheses.

Scalar functions may be nested within scalar functions or column functions, and column functions may be nested within scalar functions. See [Rules for Scalar Functions](#) (see page 555) for more information on nesting.

Following is the syntax diagram for scalar functions. This is CA Datacom/DB extension. For details, see the following:

- [Character Functions](#) (see page 565).
- [Bit-Level Functions](#) (see page 570).
- [Byte-Level Function](#) (see page 573).
- [XML Functions](#) (see page 574).



## Rules for Scalar Functions

Scalar functions may be nested within scalar functions. For example:

`DATE(TIMESTAMP(LASTCHANGED))` results in the conversion of the value in a column from a string representation of a `TIMESTAMP` into a `TIMESTAMP` value and then extracts the `DATE` from that `TIMESTAMP`.

Scalar functions may be nested within column functions. For example:

`AVG(YEAR(HIREDATE))` results in the average year all hiring was done.

Column functions may be nested within scalar functions. For example:

`YEAR(MAX(HIREDATE))` results in the last year hiring was done.

## Description

The following descriptions of the entries shown in the previous syntax diagram are listed in alphabetical order, except for the description of *expression*, described first because of its multiple occurrences in the diagram. Also, see [Rules for Scalar Functions](#) (see page 555).

### ***bit-level-functions***

See [Bit-Level Functions](#) (see page 570)

### ***byte-level-function***

See [Byte-Level Function](#) (see page 573)

### **CHAR**

Use the `CHAR` function to obtain a string representation of a date/time value. The result is a fixed-length character string. Its first argument must be a date, time, or timestamp. Its second argument is used, when the first argument is a date or time, to specify an ISO, USA, EUR, or JIS string format.

If the first argument is a date, the result has a length of 10 and is the character string representation of the date in the format specified by the second argument.

If the first argument is a time, the result has a length of eight and is the character string representation of the time in the format specified by the second argument.

If the first argument is a timestamp, the result has a length of 26 and is the character string representation of the timestamp. Do not specify a second argument (for a string format) when the first argument is a timestamp.

See [Character String Literals](#) (see page 510) for information about date, time, and timestamp formats.

**,ISO**

Specifies International Standards Organization format.

**,USA**

Specifies International USA Standard format.

**,EUR**

Specifies IBM European Standard format.

**,JIS**

Specifies Japanese Industrial Standard format.

***character-functions***

See [Character Functions](#) (see page 565)

**DATE**

Use the DATE function to obtain a date from a value. The result is a date. Its argument must be a date, timestamp, a string representation of a date, a character string of length 7, or a positive number.

For example, DATE(TIMESTAMP('1989-03-20-11.30.00') + 2 DAYS) results in a date of 1989-03-22 (in ISO or JIS format).

If its argument is a character string of length seven, it is assumed to have the form `yyyynnn` where `yyyy` is the year and `nnn` is the day within the year in the range of 001 to 366.

For example, DATE('1989079') results in a date of 1989-03-20 (in ISO or JIS format).

If the argument is a positive number, `n`, the result is the date that is `n` days after December 31, 0000.

For example, DATE(32) results in a date of 0001-02-01 (in ISO or JIS format).

**DAY**

Use the DAY function to obtain the day part of a value. The result is an integer representing a day. The sign of the result is negative only if the value of its expression is a negative duration. Its argument must be a date, timestamp, or DECIMAL(8,0) number interpreted as a date-duration.

For example, if BIRTHDATE is March 25, 1945, that is to say, 19450325, then DAY(BIRTHDATE) results in 25.

**DAYS**

Use the DAYS function to obtain an integer representation of a date. The result is an integer representing the number of days since December 31, 0000 (that is to say, January 1 is 1 day, February 1 is 32 days, and so on). The sign of the result is always positive. Its argument can be a date, timestamp, or string representation of a date.

For example, DAYS('0001-02-01') results in 32 days.

**DECIMAL**

Use the DECIMAL function to obtain a decimal representation of a numeric value. Three arguments are possible. The first argument is required. The second and third arguments (labeled as "integer" in the syntax diagram) are optional. The result is a decimal number with a precision of  $p$  and a scale of  $s$ , where  $p$  is the second argument and  $s$  is the third argument.

The first argument must be a number. If you specify a second argument, it represents the precision  $p$  and must be an integer in the range of 1 to 31. If you specify a third argument, it represents the scale  $s$  and must be an integer in the range of 0 to the  $p$  specified in the second argument. You cannot specify a third argument if you have not specified a second argument.

Omitting the third argument results in a value of 0 for the scale. Omitting the second argument results in:

- $p = 15$  if the first argument is floating-point or decimal
- $p = 11$  if the first argument is a large integer
- $p = 5$  if the first argument is a small integer.

The result can be null if the first argument can be null; the result is the null value if the first argument is null. The result is the same number that would occur if the first argument were assigned to a decimal column or variable with a precision of  $p$  and a scale of  $s$ .

If the number of significant decimal digits required to represent the whole part of the number is greater than  $p-s$ , an error occurs.

For example, if SALARY is a FLOAT column, DECIMAL(AVG(SALARY),8,2) results in the average salary being converted to a packed decimal value of xxxxxx.xx.

### DIGITS

Use the DIGITS function to obtain a fixed-length character string representation of a number. Its argument must be an integer or a decimal number. The string of digits that make up the result represent the absolute value of the argument without regard to its scale. The result therefore does not include a sign or a decimal point. Leading zeros are included in the result as necessary so that length of string equals:

- 5 if argument is a small integer
- 10 if argument is a large integer
- $p$  if argument is a decimal number with precision  $p$ .

The result can be null if the argument can be null; the result is the null value if the argument is null.

For example, if the data type of COLUMNX is DECIMAL(6,2), and if COLUMNX has a value of -7.27, then DIGITS(COLUMNX) gives '000727' as the result.

#### *(expression)*

Enter the expression which is to be the argument of the function. For more information about expressions, see [Expressions](#) (see page 527).

### FLOAT

Use the FLOAT function to obtain a floating-point representation of a number. Its argument must be a number. A double precision floating-point number is the result. The result can be null if the argument can be null; the result is the null value if the argument is null.

For example, if ACSTAFF is an INTEGER column, FLOAT(ACSTAFF)/2 results in the double precision floating-point representation of half of the value in ACSTAFF.

### HEX

Use the HEX function to obtain an hexadecimal representation of a value. A character string is the result of the function. The result can be null if the argument can be null; the result is the null value if the argument is null.

In the string of hexadecimal digits that form the result, the first two digits represent the first byte of the argument, the second two digits the second byte, and so on. If a date or time value is the argument, the result is the hexadecimal representation of the internal form of the argument.

The length of the result is twice the defined (maximum) length of the argument.

If the argument is not a varying-length string and the length of the result is less than 255, the result is a fixed-length string. Otherwise, the result is a varying-length string whose maximum length depends on the following considerations. If the argument:

- Is not a varying-length string: maximum length of the result string is the same as the length of the result.
- Is a varying-length string: maximum length of the result string is twice the maximum length of the argument.

For example, if 'ABC' is contained in CHAR column COLX, HEX(COLX) results in the fixed-length string 'C1C2C3'.

### **HOUR**

Use the HOUR function to obtain the hour part of a value. The result is an integer representing an hour. The sign of the result is negative only if the value of its argument is a negative duration. Its argument must be a time, timestamp, or a DECIMAL(6,0) number interpreted as a time-duration.

For example, if TIME1 is a timestamp of 19890203093020009900, then HOUR(TIME1) results in 9.

### **INTEGER**

Use the INTEGER function to obtain an integer representation of a number. The argument must be a number. A large integer is the result of the function. The result can be null if the argument can be null; the result is the null value if the argument is null.

The result obtained by this function is the same number that would occur if the argument were assigned to a large integer column or variable. An error occurs if the whole part of the argument is not within the range of integers.

For example, if PAYNUM is a DECIMAL(8,2) column, INTEGER(SUM(PAYNUM)+.5) results in the sum (rounded up) as an integer value.

### **LENGTH**

Use the LENGTH function to obtain the length of a value. Any value can be used as the argument. The result is a large integer. The result can be null if the argument can be null. The result is the null value if the argument is null.

The length of the argument is the result. The null indicator byte of column arguments that allow null values is not included in the length. Blanks are included in the length of strings, but the length control field of varying-length strings is not included in the length. The actual (not the maximum) length of varying-length strings is the length.

The length is the number of bytes used to represent the value as follows:

- Character string: length of the string
- Small integer: 2
- Large integer: 4
- Floating-point: 8
- Decimal numbers with precision  $p$ :  $\text{INTEGER}(p/2) + 1$
- Date: 4
- Time: 3
- Timestamp: 10
- Numeric numbers with precision  $p$ :  $p$

For example, if COLX is a VARCHAR(20) column, LENGTH(COLX) returns the actual length of the string in that column.

#### **MICROSECOND**

Use the MICROSECOND function to obtain the microsecond part of a value. The result is an integer representing a number of microseconds. The sign of the result is always positive. Its argument must be a timestamp.

For example, if TIME1 is 19890320093020109000 then, MICROSECOND(TIME1) results in 109000 microseconds.

#### **MINUTE**

Use the MINUTE function to obtain the minute part of a value. The result is an integer representing a minute. The sign of the result is negative only if the value of its argument is a negative duration. Its argument must be a time, timestamp, or a DECIMAL(6,0) number interpreted as a time-duration.

For example, if TIME1 is a timestamp of 19890203093020009900, then MINUTE(TIME1) results in 30.



**MONTH**

Use the MONTH function to obtain the month part of a value. The result is an integer representing a month. The sign of the result is negative only if the value of its expression is a negative duration. Its argument must be a date, timestamp, or DECIMAL(8,0) number interpreted as a date-duration.

For example, if BIRTHDATE is of March 25, 1945, that is to say, 19450325, then MONTH(BIRTHDATE) results in 3.

**SECOND**

Use the SECOND function to obtain the seconds part of a value. The result is an integer representing a second. The sign of the result is negative only if the value of its argument is a negative duration. Its argument must be a time, timestamp, or a DECIMAL(6,0) number interpreted as a time-duration.

For example, if TIME1 is a timestamp of 19890203093020009900, then SECOND(TIME1) results in 20.

**SUBSTR**

Use the SUBSTR function to obtain a substring of a string. Three arguments are possible: *string*, *start*, and *length*. The *string* and *start* arguments are required, but the *length* argument is optional. The result can be null if any of the arguments can be null. The result is the null value if any of the arguments are null.

**Note:** The SUBSTR function accepts mixed data strings. However, because SUBSTR operates on a strict byte-count basis with character strings, the result is not necessarily a properly formed mixed data string.

DBCS characters can be used in either GRAPHIC or CHAR with MIXED DATA. When GRAPHIC is used, there is no problem because there are no shift-out or shift-in bytes, and *start* and *length* refer to double-byte characters (not bytes). However, be aware that in CHAR MIXED DATA, the shift-out and shift-in may not balance in the result string.

You can use the VARGRAPHIC scalar function to normalize data before doing comparisons.

***string***

Refers to an expression specifying the string from which the result is derived. The *string* must be a character string or a graphic string. If *string* is a character string, the result of the function is a character string. If it is a graphic string, the result of the function is a graphic string.

A substring of the *string* argument is 0 or more contiguous characters of the *string* argument.

***,start***

Refers to an expression specifying the position of the first character of the result. The *start* argument must be a positive binary integer that is not greater than the length attribute of the *string* argument. Note that the length attribute of a varying-length string is its maximum length.

**,length**

Refers to an expression specifying the length of the result. If you specify the length argument, it must be a binary integer in the range of 0-n, where n is the length attribute of the string argument minus the start argument plus 1, with the exception that n must not be the integer constant 0.

If you explicitly specify the length argument, the string argument is effectively padded on the right with the necessary number of blank characters so that the specified substring of the string argument always exists. The length argument has a default of the number of characters from the character specified by the start argument to the last character of the string argument, but if the string argument is a varying-length string with a length that is less than the start argument, the default is 0 and the result is the empty string.

If you explicitly specify the length argument to be an integer constant of less than 255, the result is a fixed-length string. If you do not explicitly specify the length argument but the string argument is a fixed-length string and the start argument is an integer constant, the result is a fixed-length string. In all other cases, the result is a varying-length string with a maximum length that is the same as the length attribute of the string argument.

If the string argument is a fixed-length string, omitting the length argument is an implicit specification of

`LENGTH(string) - start + 1`

If the string argument is a varying-length string, omitting the length argument is an implicit specification of either

`LENGTH(string) - start + 1`

or 0, whichever is greater.

For example, if `FIRSTNAME` is a `VARCHAR(20)` column, `SUBSTR(FIRSTNAME,1,1)` results in a fixed-length string value containing the first character of the value in `FIRSTNAME`.

**TIME**

Use the TIME function to obtain the time from a value. The result is a time. Its argument must be a time, timestamp, a string representation of a time. For example if TIME1 is 10890320093020, then TIME(TIME1) results in time 9.30.20 (in ISO or EUR format).

**TIMESTAMP**

Use the TIMESTAMP function to obtain a timestamp from a value or a pair of values. The result is a timestamp. If only one argument is given, it must be a timestamp, a string representation of a timestamp, a character string of length 8, or a character string of length 14. If the value is a character string of length 8, it is assumed to be a STORE CLOCK value representation of a timestamp. If the value is a character string of length 14, it must be in the form `yyyymmddhhmmss` where `yyyy` is the year, `mm` the month, `dd` the day, `hh` the hours, `mm` the minutes, and `ss` the seconds.

If a second optional argument is specified, the second argument must be a time or a string representation of a time, and the first argument must be a date or a string representation of a date.

For example, `TIMESTAMP(CURRENT DATE, '11.30.00')` results in the timestamp 11:30 today.

**VALUE**

Use the VALUE function to substitute a value for the null value. The arguments' data types must be compatible. Because character strings are not converted to date/time values, all arguments must be dates if any argument is a date. Similarly, all arguments must be times if any argument is a time, all must be timestamps if any are timestamps, and all must be character strings if any are character strings.

Arguments are evaluated in the order of specification. The result of the function is equal to the first argument that is not null. The result is nullable. That is, it is not NOT NULL. The result is the null value only if all arguments are null.

The result is defined as equal to an argument because that argument is converted or extended, if necessary, in order to conform to the data type of the function. The data type of the result is derived from the data types of the specified arguments as follows:

***strings:***

If any argument is a varying-length string, the result is a varying-length string whose maximum length is equal to the longest string that can result from the application of the function.

If all arguments are fixed-length strings, the result is a fixed-length string whose length is equal to the longest string that can result from the application of the function.

***date/time values:***

The result is a date if the arguments are dates, times if the arguments are times, or timestamps if the arguments are timestamps.

***numbers:***

If the arguments are numbers, the result is the numeric data type that would occur if all arguments were part of a single arithmetic expression. If that data type is decimal, it has a precision of  $p$  and a scale of  $s$ . Therefore,  $s$  is the largest result scale of any argument and  $p$  is the minimum of 31 and  $s + n$ , where  $n$  is the largest integral part result of any argument. Conversion errors are possible if the sum of  $s + n$  is greater than 31.

For example, `SALARY + VALUE(COMMISSION,0)` results in the sum of `SALARY` and `COMMISSION` if `COMMISSION` is not the null value, or the sum of `SALARY` and 0 if `COMMISSION` contains the null value.

**VARGRAPHIC**

Use the VARGRAPHIC function to convert SBCS (Single-Byte Character Set) and MIXED strings to VARGRAPHIC.

In SQL, string comparisons are strictly byte-for-byte, so predicates involving MIXED strings may not give the desired results. For instance, a string with the SBCS version of XYZ does not compare as equal to a string with a Shift-Out, then the DBCS version of XYZ, then a Shift-In. You can use the VARGRAPHIC scalar function to normalize both strings into VARGRAPHIC data types before doing the comparison.

When SBCS strings are converted, each SBCS character is paired with a X'42' to form the double-byte character. In MIXED strings, the SBCS characters are converted, and Shift-In and Shift-Out characters are removed. FOR BIT DATA operands are not allowed.

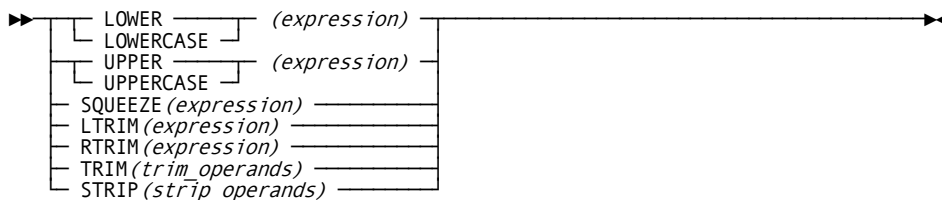
**YEAR**

Use the YEAR function to obtain the year part of a value. The result is an integer representing a year. The sign of the result is negative only if the value of its expression is a negative duration. Its argument must be a date, timestamp, or DECIMAL(8,0) number interpreted as a date-duration.

For example, if BIRTHDATE is March 25, 1945, that is to say, 19450325, then YEAR(BIRTHDATE) results in 1945.

## Character Functions

Following is the syntax diagram for character-level scalar functions:



**LOWER(*expression*) or LOWERCASE(*expression*)**

This character function returns a copy of the result of the input expression that has been converted to lowercase characters. The input expression must resolve to the character data type and cannot consist of bit, mixed, or Katakana data. Bit and mixed data are rejected. Katakana data is processed, producing an unpredictable result string. The data type, length, and nullability of the result matches that of the input.

Following is an example of using the LOWER scalar function:

```
SELECT LOWER(last_name) FROM customer_names
```

**Given input of**

```
' Smith '
```

**Resulting output is**

```
' smith '
```

**UPPER(*expression*) or UPPERCASE(*expression*)**

This character function returns a copy of the result of the input expression that has been converted to uppercase characters. The input expression must resolve to the character data type and cannot consist of bit, mixed, or Katakana data. Bit and mixed data are rejected. Katakana data is processed, producing an unpredictable result string. The data type, length, and nullability of the result matches that of the input.

Following is an example of using the UPPER scalar function:

```
SELECT UPPER(last_name) FROM customer_names
```

**Given input of**

```
' Smith '
```

**Resulting output is**

```
' SMITH '
```

**SQUEEZE(expression)**

This character function returns a copy of the result of the input expression that has had both leading and trailing *white space* (blanks, nulls, new lines (line feeds), carriage returns, horizontal tabs and form feeds (vertical tabs)) removed and has had any embedded *white space* converted to blanks. The input expression must resolve to the variable-length character (VARCHAR) data type and cannot consist of bit or mixed data. Katakana strings are processed as if they were EBCDIC. The data type and nullability of the result matches that of the input. The length of the result is the *squeezed* length.

Following is an example of using the SQUEEZE scalar function:

```
SELECT SQUEEZE(last_name) FROM customer_names
```

**Given input of**

```
' Smith '
```

**Resulting output is**

```
'Smith'
```

**LTRIM(expression)**

This character function returns a copy of the result of the input expression that has had leading blanks removed. The input expression must resolve to the character data type and may not consist of bit or mixed data. Katakana strings are processed as if they were EBCDIC. The result is VARCHAR with nullability matching that of the input. The length of the result is the trimmed (shortened) length.

Following is an example of using the LTRIM scalar function:

```
SELECT LTRIM(last_name) FROM customer_names
```

**Given input of**

```
' Smith '
```

**Resulting output is**

```
'Smith '
```

**RTRIM(*expression*)**

This character function returns a copy of the result of the input expression that has had trailing blanks removed. The input expression must resolve to the character data type and cannot consist of bit or mixed data. Katakana strings are processed as if they were EBCDIC. The result is VARCHAR with nullability matching that of the input. The length of the result is the trimmed (shortened) length.

Following is an example of using the RTRIM scalar function:

```
SELECT RTRIM(last_name) FROM customer_names
```

**Given input of**

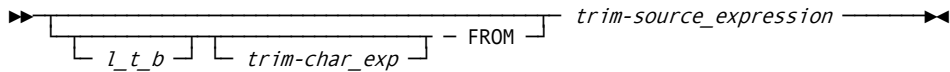
' Smith '

**Resulting output is**

' Smith'

**TRIM(*trim\_operands*)**

The TRIM character function removes extraneous characters from the start and/or end of a character string. The *trim-source\_expression* (see following) must resolve to a CHAR, VARCHAR, or GRAPHIC data type and must be compatible with the data type of the *trim-char\_expression* as shown in the following description. Katakana strings are processed as if they were EBCDIC. For CHAR and VARCHAR input, the result is VARCHAR. For GRAPHIC input, the result is VARGRAPHIC. Nullability of the result matches that of the input. The length of the result is the trimmed (shortened) length.



***l\_t\_b***

This parameter specifies where to remove unwanted characters.

Specify LEADING or L if you want to remove unwanted characters from the start of the input source string.

Specify TRAILING or T if you want to remove unwanted characters from the end of the input source string.

Specify BOTH or B if you want to remove unwanted characters from both the start and the end of the input source string. The default is BOTH.



**trim-char\_exp**

This trim character expression parameter specifies the character to be removed. The default is the blank character.

**trim-source\_expression**

This parameter supplies the input string to be operated upon.

Following is an example of using the TRIM function:

```
SELECT TRIM(BOTH ' ' FROM last_name) FROM customer_names
```

**Given input of**

```
' Smith '
```

**Resulting output is**

```
'Smith'
```

**STRIP(strip\_operands)**

The STRIP character function, provided for DB2 syntax compatibility, operates identically to the TRIM function described above in that STRIP removes extraneous characters from the start and/or end of a character string. For details, see the previously given TRIM description.

►► *strip-source\_expression* [ , *l\_t\_b* [ , *strip-char\_exp* ] ]

**strip-source\_expression**

This parameter supplies the input string to be operated upon.

**l\_t\_b**

This parameter specifies where to remove unwanted characters.

Specify LEADING or L if you want to remove unwanted characters from the start of the input source string.

Specify TRAILING or T if you want to remove unwanted characters from the end of the input source string.

Specify BOTH or B if you want to remove unwanted characters from both the start and the end of the input source string. The default is BOTH.

**, strip-char\_exp**

This strip character expression parameter specifies the character to be removed. The default is the blank character.

Following is an example of using the STRIP function:

```
SELECT STRIP(last_name, BOTH, ' ') FROM customer_names
```

**Given input of**

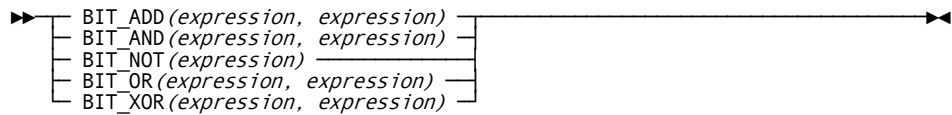
' Smith '

**Resulting output is**

'Smith'

## Bit-Level Functions

Following is the syntax diagram for bit-level scalar functions:



The following rules define the input and output of each bit-level function, that is to say, all of these rules apply to every function.

- The expression that represents each parameter must resolve to INTEGER, SMALLINT, CHAR, or VARCHAR data. In this case, CA Datacom operates on character values of any length and rejects input parameters containing mixed data (mixed single-byte/double-byte). Though not required, we recommend using the FOR BIT DATA specification in the CREATE TABLE statement for columns that are going to use bit-level functions.
- For functions that receive two parameters, character and numeric parameters cannot be used in the same function call (numeric means INTEGER or SMALLINT in this context).
- The data type, length, and nullability of the result matches that of the input if there is either only one parameter or all parameters are of identical data type, length, and nullability. The attributes of the result are otherwise determined as follows:
  - The result is nullable (that is, it is *not* NOT NULL) if either of the parameters are nullable.
  - The length of the result matches the length of the longer parameter.
  - If both parameters are numeric, the result is INTEGER if at least one of the parameters is INTEGER, or SMALLINT if neither of the parameters is INTEGER.
  - If both parameters are character (VARCHAR is character) the data type of the result matches that of the first operand. This allows you to control the data type of the result when the data types of the parameters differ.

**BIT\_ADD(expression, expression)**

This bit-level function returns the logical sum of the results of the input expressions. The logical sum is obtained by using logical addition and discarding any arithmetic overflow that is generated.

Following is an example of using BIT\_ADD to construct a web address for a computer that is being turned on.

```
SELECT BIT_ADD(web_addrs_company_prefix,web_addrs_assigned_suffix) FROM
web_addrses
```

**Given input of**

A web\_addrs\_company\_prefix of 127.255.0.0 as an integer representation or 0x7FFF0000 in C-style notation and a web\_addrs\_assigned\_suffix of 0.0.255.254 as an integer representation or 0x0000FFEE in C-style notation

**Resulting output is**

An integer representation of 127.255.255.254 or 0x7FFFFFFE in C-style notation

**BIT\_AND(expression, expression)**

This bit-level function returns the logical AND of the results of the input expressions consisting of a (1) bit in the result for every input bit-pair of (1,1) and a (0) bit for all other bit-pair value combinations.

**Note:** A single 1 or 0 digit inside parentheses, that is a (1) or a (0), is used here to represent a single bit value. The term *bit-pair* refers to the value of a bit taken from the first input parameter and that of a bit taken from the corresponding position in the second input parameter. For example, a bit-pair of (1,0) refers to a 1 bit somewhere in the first parameter and a 0 bit in the corresponding position of the second parameter.

Following is an example of using BIT\_AND to find what area of the web a browser is pointed to (that is, get the high-level portion of the web address).

```
SELECT BIT_AND(:bits_to_extract, web_addrs), user_name FROM web_addrses
```

**Given input of**

A bits\_to\_extract of 127.255.0.0 as an integer representation or 0x7FFF0000 in C-style notation, used to denote the bits you want to extract from the web address (here, bits\_to\_extract has a colon (:) in front of it to show it can be a host variable from a theoretical user program) and a web\_addrs of 127.255.1.1 as an integer representation or 0x7FFF0101 in C-style notation

**Resulting output is**

An integer representation of 127.255.0.0 or 0x7FFF0000 in C-style notation

**BIT\_NOT(expression)**

This bit-level function returns the logical NOT (the complement) of the results of the input expression. Each (1) bit in the input parameter becomes a (0) bit in the result, and each (0) bit becomes a (1) bit.

**Note:** A single 1 or 0 digit inside parentheses, that is a (1) or a (0), is used here to represent a single bit value.

Following is an example of using BIT\_NOT to find out which status flags are not set.

```
SELECT BIT_NOT(status_bits) from system_status
```

**Given input of**

A status\_bits of 0x7F010101 in C-style notation

**Resulting output is**

0x80FEFEFE in C-style notation

**BIT\_OR(expression, expression)**

This bit-level function returns the logical OR of the results of the input expressions. Each input bit-pair of (0,0) becomes a (0) bit in the result, and all other bit-pair value combinations become (1) bits.

**Note:** A single 1 or 0 digit inside parentheses, that is a (1) or a (0), is used here to represent a single bit value. The term *bit-pair* refers to the value of a bit taken from the first input parameter and that of a bit taken from the corresponding position in the second input parameter. For example, a bit-pair of (1,0) refers to a 1 bit somewhere in the first parameter and a 0 bit in the corresponding position of the second parameter.

Following is an example of using BIT\_OR to construct a web address for a computer being turned on.

```
SELECT BIT_OR(web_addrs_company_prefix, web_addrs_assigned_suffix) from  
web_addrses
```

**Given input of**

A web\_addrs\_company\_prefix of 127.255.0.0 in an integer representation or 0x7FFF0000 in C-style notation and a web\_addrs\_assigned\_suffix of 0.0.255.254 in an integer representation or 0x0000FFFE in C-style notation

**Resulting output is**

An integer representation of 127.255.255.254 or 0x7FFFFFFE in C-style notation.

**BIT\_XOR(expression, expression)**

This bit-level function returns the logical exclusive-OR of the results of the input expressions. Each bit-pair containing *exactly* one (1) bit becomes a (1) bit in the result. That is to say, each input bit-pair of (0,1) or (1,0) becomes a (1) bit in the result, and all other bit-pair value combinations, (0,0) or (1,1), become (0) bits.

**Note:** A single 1 or 0 digit inside parentheses, that is a (1) or a (0), is used here to represent a single bit value. The term *bit-pair* refers to the value of a bit taken from the first input parameter and that of a bit taken from the corresponding position in the second input parameter. For example, a bit-pair of (1,0) refers to a 1 bit somewhere in the first parameter and a 0 bit in the corresponding position of the second parameter.

Following is an example of using BIT\_XOR to find out which system status flags do not match a required value.

```
SELECT BIT_XOR(status_bits, required_status_bits) from system_status
```

**Given input of**

A status\_bits of 0x7F010101 in C-style notation and a required\_status\_bits of 0x7F111110 in C-style notation

**Resulting output is:**

0x00101011 in C-style notation, giving the bits that do not match

## Byte-Level Function

Following is the syntax diagram for byte-level scalar functions:

►► INEXTRACT(*expression, expression*) ◄◄

**INEXTRACT(expression, expression)**

This byte-level function is used to extract the binary value of a single byte of a data value. The function returns an INTEGER result, hence the name INEXTRACT (integer extract). The first parameter, meaning the first expression in INEXTRACT(expression, expression), supplies the data value, and the second parameter (the second expression) specifies which byte is wanted.

For example, if variable *web\_addrs* in a C-language program contains the value (in C-style notation) 0x11223344, then INEXTRACT(web\_addrs, 2) returns the value 0x22 in C-style notation or 34 in decimal-style notation, extracted from the second byte from the high-order (left-hand) side of the value, returning an INTEGER result.

#### The first expression

Representing the first parameter must resolve to INTEGER, SMALLINT, CHAR (non-mixed data only, that is to say, mixed DBCS/SBCS only), or VARCHAR (non-mixed data only). The length of the first parameter is limited only to the SQL-imposed limits for each data type. Though not required, we recommend use of the FOR BIT DATA specification in the CREATE TABLE statement for columns that are going to use byte-level functions.

#### The second expression

Representing the second parameter must resolve to INTEGER or SMALLINT data.

Following is an example of using INTEXTRACT to get a single node of a web address:

```
SELECT INTEXTRACT(web_addrs, 2) FROM web_addresses
```

#### Given input of

A web\_addrs of 0x11223344 in C-style notation.

#### Resulting output is

An integer containing 0x00000022 in C-style notation or 34 in decimal-style notation.

## XML Functions

### XML Overview

The Extensible Markup Language (XML) functions described in the following sections allow you to externalize relational data as XML data. The CA Datacom/DB implementation of XML includes support for the following XML functions:

- XMLELEMENT (see following description)
- XMLATTRIBUTES (valid within use of XMLELEMENT only)
- XMLFOREST (see [XMLFOREST](#) (see page 576))
- XMLSERIALIZE (see [XMLSERIALIZE](#) (see page 578))
- XMLCONCAT (see [XMLCONCAT](#) (see page 579))

The XMLELEMENT, XMLATTRIBUTES, XMLFOREST, and XMLCONCAT functions operate on relational or XML data to produce XML output. The XMLSERIALIZE function is an ANSI-compliant method for returning XML values into user applications that expect string result types such as VARCHAR. An XML value is defined as a well-formed XML document or a document fragment consisting of well-formed XML content.

### Descriptions of Functions

Following are descriptions of the supported XML functions.

## XMLELEMENT

This function builds and returns an XML value given the following:

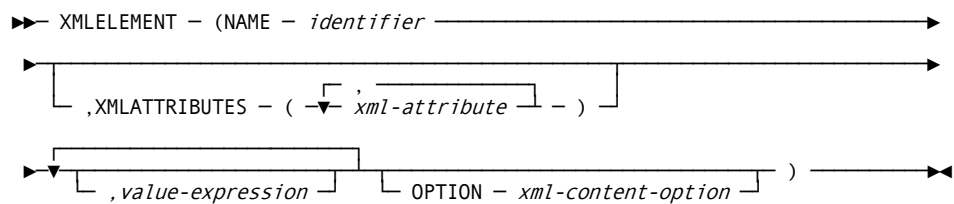
- An XML element name,
- An optional list of XML attributes, and
- An optional list of values as the content of the new element.

**Note:** The XML namespace declaration is not supported.

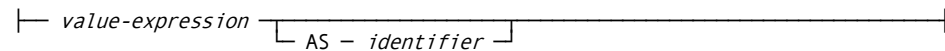
## XMLATTRIBUTES

This function builds a list of attributes to be included in the XML element generated by the XMLELEMENT function. XMLATTRIBUTES is valid only within an XMLELEMENT function.

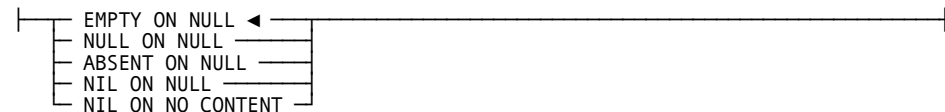
Following is the syntax diagram for XMLELEMENT and XMLATTRIBUTE:



Expansion of Where *xml-attribute* is as follows



Expansion of Where *xml-content-option* is as follows



The *xml-content-option* defines the XML value that is produced when the *value-expression* list, representing the content of the element, is either missing or evaluates entirely to NULLs. If no content is supplied, then any supplied ON NO CONTENT specification is applied. If content is supplied but every supplied item evaluates to NULL, the ON NULL option applies. In the XMLELEMENT function, EMPTY ON NULL is the default. In XMLFOREST, NULL ON NULL is the default. In all other cases, an element is generated using whatever non-NULL values were supplied.

### NULL ON NULL

Produces a NULL rather than an element.

### EMPTY ON NULL

Produces an element with no content (subject to change or removal).

**ABSENT ON NULL**

Produces a zero-length result (subject to change or removal).

**NIL ON NULL and NIL ON NO CONTENT**

Produce elements with no content but containing attributes that read "nil=true" when serialized.

The following example produces a Customer element for each customer, with customer number and name attributes:

```
SELECT XMLSERIALIZE(CONTENT
                    XMLELEMENT(NAME "Customer",
                               XMLATTRIBUTES(CustNo,
                                               SurName as LastName,
                                               FirstName)
                               )
                    AS VARCHAR(200)) AS "CustomerList"
FROM customers;
```

**Note:** The XMLSERIALIZE function (see [XMLSERIALIZE](#) (see page 578)) converts the XML output of XMLELEMENT to VARCHAR in this example.

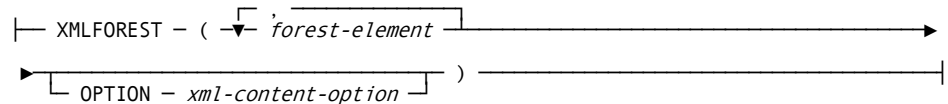
The result of the previous example follows:

```
CustomerList
VARCHAR(200)
-----
<-Customer CustNo="000001" LastName="Sturlasson" FirstName="Snorri"></Customer>
<-Customer CustNo="000002" LastName="Skallagrimsson"
FirstName="Eigil"></Customer>
-----
```

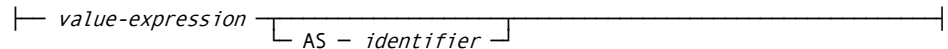
**XMLFOREST**

This function returns an XML value consisting of a forest, or collection, of XML elements, given a list of "forest elements." Each forest element generates an XML element using the referenced column name or, if provided, the forest element name (the *identifier* in the syntax diagram that follows) as the XML element name and the forest element value (the *value expression* in the following diagram) as the element content.

Following is the syntax diagram for XMLFOREST:

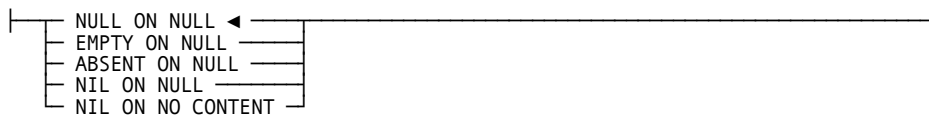


Expansion of Where forest-element is as follows





Expansion of Where *xml-content-option* is as follows



The *xml-content-option* in the XMLFOREST function is applied separately to each *forest-element*. In XMLFOREST, EMPTY ON NULL is the default. For a further description of the *xml-content-option*, see the paragraph following the XMLELEMENT syntax diagram in [XMLATTRIBUTES](#) (see page 575).

The following example generates a "Vendor" element for each vendor. The name of the vendor is used as an attribute, and two sub-elements are created from columns `contactName` and `contactPhone` using the XMLFOREST function.

```
SELECT vendorId,
       XMLSERIALIZE(CONTENT
                    XMLELEMENT(NAME "Vendor",
                               XMLATTRIBUTES(v.vendorDBA AS DBA),
                               XMLFOREST (v.contactName,
                                           v.contactPhone AS phone)
                               )
                    AS VARCHAR(300)) AS "vendorContacts"
FROM vendors v;
```

The result of the previous example follows:

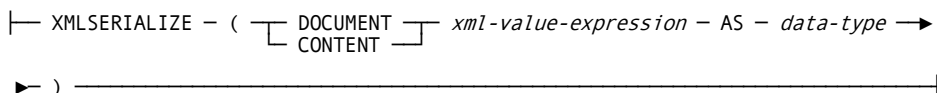
```
vendorId  vendorContacts
INTEGER   VARCHAR(300)
-----
1  <-Vendor DBA="Best Office Supplies">
    <-contactName>Joseph Dudely<-/contactName>
    <-phone>123-456-7890<-/phone>
  <-/Vendor>
2  <-Vendor DBA="Widgets R Us">
    <-contactName>Mary Doeright<-/contactName>
    <-phone>123-456-1313<-/phone>
  <-/Vendor>
-----
```

## XMLSERIALIZE

This function converts an XML value into the corresponding representation in an alternative character-string data-type. XMLSERIALIZE converts XML types to string types for export to user host-variables.

**Note:** When XMLSERIALIZE is used to externalize floating point data, 15 is the maximum number of significant digits produced.

Following is the syntax diagram for XMLSERIALIZE:



### DOCUMENT

Specifies that the result of the *xml-value-expression* is a validly formed XML document.

### CONTENT

Specifies well-formed XML content.

### *xml-value-expression*

Any expression whose result is an XML value, such as XMLELEMENT (see [XMLELEMENT](#) (see page 575)), XMLCONCAT (see [XMLCONCAT](#) (see page 579)), or XMLFOREST (see [XMLFOREST](#) (see page 576)).

### *data-type*

Must specify some character-string type.

The following example is a repetition of our XMLELEMENT example and produces a Customer element for each customer, with customer number and name attributes:

```

SELECT XMLSERIALIZE(CONTENT
                    XMLELEMENT(NAME "Customer",
                               XMLATTRIBUTES(CustNo,
                                               surName as LastName,
                                               FirstName)
                               )
                    AS VARCHAR(200)) AS "CustomerList"
FROM customers
    
```

The result of the previous example follows:

```
CustomerList
VARCHAR(200)
-----
<-Customer CustNo="000001" LastName="Sturlasson" FirstName="Snorri"><-/Customer>
<-Customer CustNo="000002" LastName="Skallagrimsson"
FirstName="Eigil"><-/Customer>
-----
```

## XMLCONCAT

This function returns an XML value that is the concatenation of a list of XML values.

Following is the syntax diagram for XMLCONCAT:

```
| XMLCONCAT - ( ( xml-value-expression ) )
```

### *xml-value-expression*

Any expression whose result is an XML value, such as XMLELEMENT (see [XMLELEMENT](#) (see page 575)), XMLCONCAT (see [XMLCONCAT](#) (see page 579)), or XMLFOREST (see [XMLFOREST](#) (see page 576)).

The following example (a repetition of the XMLELEMENT example) produces a Customer element for each customer, with customer number and name attributes:

```
SELECT XMLSERIALIZE(CONTENT
                XMLCONCAT(XMLELEMENT(NAME "contact",v.contactName),
                XMLELEMENT(NAME "phone",v.contactPhone)
                )
                AS VARCHAR(300)) AS "vendorContacts"
FROM vendors v;
```

The result of the previous example follows:

```
vendorContacts
VARCHAR(300)
-----
<-contact>Joeseeph Dudely<-/contact><-phone>123-456-7890<-/phone>
<-contact>Mary Doeright<-/contact><-phone>123-456-1313<-/phone>
-----
```



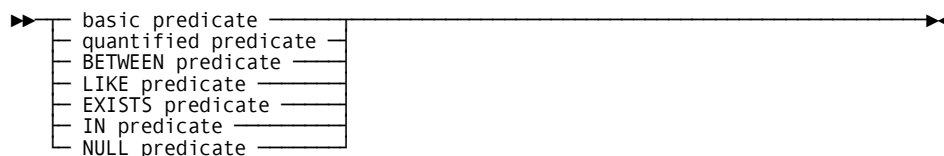
# Chapter 26: Predicates

---

A predicate specifies a condition that is "true" or "false" about a given row or group.

**Note:** In SQL, string comparisons are strictly byte-for-byte, so predicates involving MIXED strings may not give the desired results. For instance, a string with the SBCS version of XYZ does not compare as equal to a string with a Shift-Out, then the DBCS version of XYZ, then a Shift-In. You can use the VARGRAPHIC scalar function to normalize both strings into VARGRAPHIC data types before doing the comparison.

Following is the syntax diagram for a predicate:



All values specified in a predicate must be compatible.

The following sections discuss branches of the predicate's general form.

## Basic Predicate

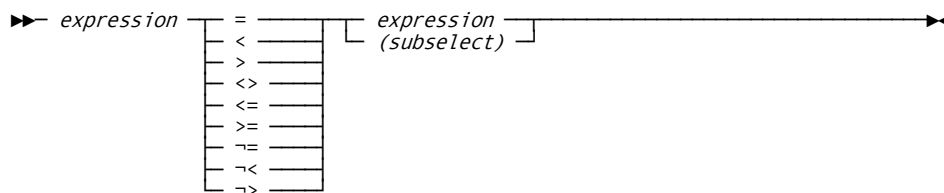
A basic predicate is used to compare two values. The format of a basic predicate is:

- An expression followed by a comparison operator and another expression.
- An expression followed by a comparison operator and a subselect. The subselect cannot be preceded by ANY, ALL, or SOME. A subselect is a form of the SELECT command. A subselect in a basic predicate must not return more than one value.

Following is the syntax diagram for a basic predicate:

**Note:** The following are CA Datacom/DB extensions:

- $\rightarrow =$  is a CA Datacom/DB extension.
- $\rightarrow <$  is a CA Datacom/DB extension.
- $\rightarrow >$  is a CA Datacom/DB extension.



**Description*****expression***

Specify an expression. For more information about expressions, see [Expressions](#) (see page 527).

***(subselect)***

Specify a subselect. The subselect must be enclosed by parentheses. For more information about the subselect see Subselect.

The result of the comparison is either true or false. The following table shows the results for each comparison operator when comparing the values x and y.

<b>Predicate</b>	<b>Is true <i>only</i> if...</b>
x = y	x is equal to y
x < y	x is less than y
x > y	x is greater than y
x <> y	x is not equal to y
x <= y	x is less than or equal to y
x >= y	x is greater than or equal to y
x != y	x is not equal to y
x -< y	x is not less than y
x -> y	x is not greater than y

The !=, -< and -> comparison operators are CA Datacom/DB extensions to SQL.

**Examples**

Some examples of basic predicates are:

**Example 1:** This example specifies that the employee number must be equal to the literal value 671.

```
EMPNO = '671'
```

**Example 2:** This example specifies that the salary must be less than \$20,000.

```
SALARY < 20000
```

**Example 3:** This example specifies that the quantity must not be equal to the value of the host-variable, VAR1.

```
QUANTITY <> :VAR1
```

**Example 4:** This example specifies that the salary must be greater than the average salary for all employees.

```
SALARY > (SELECT AVG(SALARY)
          FROM EMP)
```

## Quantified Predicate

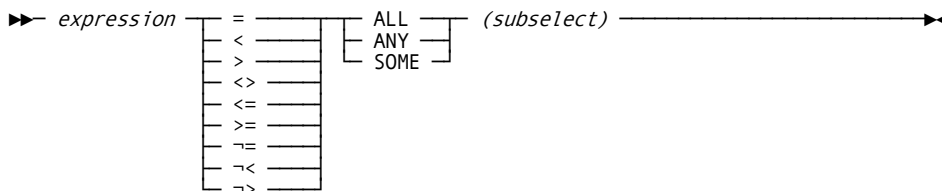
Use the quantified predicate to compare a value with a collection of values.

A quantified predicate has the same form as a basic predicate except the second operand is a subselect preceded by ANY, ALL, or SOME. The subselect may return any number of values.

Following is the syntax diagram for a quantified predicate:

**Note:** The following are CA Datacom/DB extensions:

- $\neq$  is a CA Datacom/DB extension.
- $\rightarrow<$  is a CA Datacom/DB extension.
- $\rightarrow>$  is a CA Datacom/DB extension.



**Description**

***expression***

Specify an expression. For more information about expressions, see [Expressions](#) (see page 527).

***(subselect)***

Specify a subselect. The subselect must be enclosed by parentheses. For more information about the subselect see Subselect.

**ALL**

The result of the predicate is true if the subselect returns no values or if the specified relationship is true for every value returned by the subselect.

The result is false if the specified relationship is false for at least one value returned by the subselect.

**ANY**

The result of the predicate is true if the specified relationship is true for at least one value returned by the subselect.

The result is false if the subselect returns no values or if the specified relationship is false for every value returned by the subselect.

**SOME**

The result of the predicate is true if the specified relationship is true for at least one value returned by the subselect.

The result is false if the subselect returns no values or if the specified relationship is false for every value returned by the subselect.

**Example**

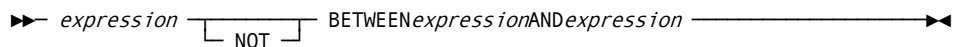
The following example of a quantified predicate specifies that value of PAY must be greater than or equal to each SALARY value returned by the SELECT. The result is true if that is the case. However, if the value of PAY is less than one of the values returned by the SELECT, then the result is false.

```
PAY >= ALL (SELECT SALARY
            FROM EMP)
```

## BETWEEN Predicate

Use the BETWEEN predicate to compare a value with a range of values.

Following is the syntax diagram for a BETWEEN predicate:.





**Description*****expression***

Specify an expression. For more information about expressions, see [Expressions](#) (see page 527).

**BETWEEN**

Introduces the range of values for the comparison.

**AND**

Joins the lower and upper limits of the value range.

**NOT**

A keyword. Use NOT to specify values that are not within the lower and upper limits of the value range.

The following table shows BETWEEN predicates and the search conditions to which they are equivalent.

The BETWEEN predicate:	is equivalent to the search condition:
value-1 BETWEEN value-2 AND value-3	value-1 >= value-2 AND value-1 <= value-3
value-1 NOT BETWEEN value-2 AND value-3	NOT (value-1 BETWEEN value-2 AND value-3)

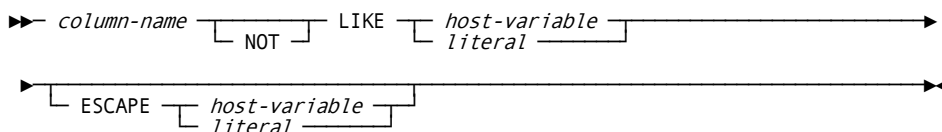
**Example**

The result of the following BETWEEN predicate is true only if the value of SALARY is greater than or equal to \$25,000, or is less than or equal to \$50,000. If the value of SALARY is not within the specified range, the result of this BETWEEN predicate is false.

```
SALARY BETWEEN 25000 AND 50000
```

## LIKE Predicate

Use the LIKE predicate to search for strings that have a certain pattern. The pattern is specified by a string in which the underscore and percent sign have special meaning. Following is the syntax diagram for the LIKE predicate:



### **Description**

#### ***column-name***

Specify the name of a column in a table or view. The column-name can be any expression that has a character string data type.

#### **NOT**

A keyword. Use NOT to specify values that do not match the comparison.

#### **LIKE *host-variable***

Specify a host-variable. The host-variable must identify a variable that is described in the program under the rules for declaring string host-variables. (See Host Variables for more information.) The host-variable must contain data of the same type as the specified column.

Specify a pattern for the comparison as part of the host-variable. The pattern must contain character data. Use the percent sign (%) in your pattern to indicate a substring of zero or more characters. Use the underscore (\_) in your pattern to indicate a single character.

See the examples at the end of this section for use of the percent sign and underscore as substring specifiers.

**Note:** When you specify a pattern (such as xxx%) that provides the prefix of the desired value, the Compound Boolean Selection Facility performs an index scan, which gives a performance gain. Any other use of a pattern or the use of a specific value causes a data scan, which is slower than an index scan.

#### **LIKE *literal***

Specify a literal, and specify a pattern for the comparison as part of the literal. The pattern must contain character data. Use the percent sign (%) in your pattern to indicate a substring of zero or more characters. Use the underscore (\_) in your pattern to indicate a single character.

See the examples at the end of this section for use of the percent sign and underscore as substring specifiers.

**Note:** When you specify a pattern (such as xxx%) that provides the prefix of the desired value, the Compound Boolean Selection Facility performs an index scan, which gives a performance gain. Any other use of a pattern or the use of a specific value causes a data scan, which is slower than an index scan.

**ESCAPE *host-variable***

Specify a host-variable. The host-variable must identify a variable that is described in the program under the rules for declaring string host-variables. (See [Host Variables](#) (see page 520) for more information.) The host-variable must contain data of the same type as the specified column.

Specify an escape character as part of the host-variable. An escape character is a valid character that is one byte in length. Use the escape character when you want to include the percent sign (%) or the underscore(\_) as part of the pattern (as previously discussed in the description of LIKE host-variable).

When an escape character is used in the pattern of the LIKE predicate, it must be followed by the percent character (%), the underscore character (\_), or another escape character. The word *character* in this context refers to either a Single-Byte Character Set (SBCS) or Double-Byte Character Set (DBCS) character. Any other use of the escape character in the pattern of the LIKE predicate is invalid.

See the examples at the end of this section for use of the escape character.

**ESCAPE *literal***

Specify a literal, and specify an escape character as part of the literal. An escape character is a valid character that is one byte in length. Use the escape character when you want to include the percent sign (%) or the underscore(\_) as part of the pattern (as previously discussed in the description of LIKE literal).

When an escape character is used in the pattern of the LIKE predicate, it must be followed by the percent character (%), the underscore character (\_), or another escape character. The word *character* in this context refers to either a Single-Byte Character Set (SBCS) or Double-Byte Character Set (DBCS) character. Any other use of the escape character in the pattern of the LIKE predicate is invalid.

See the examples at the end of this section for use of the escape character.

If the column contains character data, the terms "character," "percent sign," and "underscore" in the following discussion refer to EBCDIC (Single-Byte Character Set (SBCS)) characters. If the column contains GRAPHIC data, those terms refer to Double-Byte Character Set (DBCS) characters.

This predicate is best explained by examples. The following description is intended for those who require a rigorous definition.

1. Let *x* denote a value of the column.
2. Let *y* denote the string specified by the second operand.

3. The string *y* is interpreted as a sequence of the minimum string of substring specifiers such that each character of *y* is part of exactly one substring specifier. A substring specifier is:
  - An underscore (`_`)
  - A percent sign (`%`)
  - Any nonempty sequence of characters other than an underscore or a percent sign
4. The result is either true or false.
5. The result is true if *x* and *y* are both empty strings or there exists a partitioning of *x* into substrings such that:
  - a. A substring of *x* is a sequence of zero or more continuous characters and each character of *x* is part of exactly one substring.
  - b. If the *n*th substring specifier is an underscore, the *n*th substring of *x* is any single character.
  - c. If the *n*th substring specifier is a percent sign, the *n*th substring of *x* is any sequence of zero or more characters.
  - d. If the *n*th substring specifier is neither an underscore nor a percent sign, the *n*th substring of *x* is equal to that substring specifier and has the same length as that substring specifier.
  - e. The number of substrings of *x* is the same as the number of substring specifiers.

Predicate *x* NOT LIKE *y* is equivalent to search condition NOT(*x* LIKE *y*).

If MIXED data is in effect, the column identified by column-name may contain Double-Byte Character Set (DBCS) characters, as may the host-variable or string literal. In that case, the special characters in *y* are interpreted as follows:

- An EBCDIC (Single-Byte Character Set (SBCS)) character underscore refers to one EBCDIC character.
- A double-byte underscore refers to one DBCS character.
- A percent sign, either EBCDIC or double-byte, refers to any number of characters of any type, either EBCDIC or double-byte.



The predicate evaluates to true *only* if at least one row matches the conditions specified in the subselect.

The predicate evaluates to false if no row matches the conditions specified in the subselect.

**Note:** The subselect does not return a value. The result of the predicate cannot be unknown.

**Example**

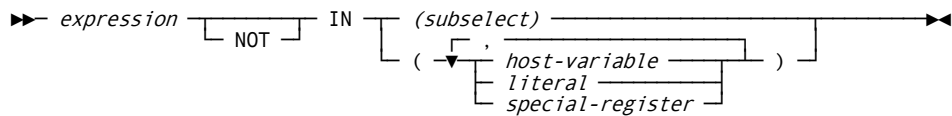
This EXISTS predicate is true only if at least one row of the result table contains a value greater than \$150,000 for the SALARY column. The result is false if no value for SALARY is greater than \$150,000.

```
EXISTS (SELECT *
        FROM EMP
        WHERE SALARY > 150000)
```

## IN Predicate

You use the IN predicate to compare a value with a collection of values. Following is the syntax diagram of the IN predicate:

**Note:** The special-register is a CA Datacom/DB extension. See [Special Registers](#) (see page 533).



**Description**

**expression**

Specify an expression. For more information about expressions, see [Expressions](#) (see page 527).

**NOT**

A keyword. Use NOT to specify that only values that do not match the comparison be selected.

**IN**

Introduces the collection of values used in the comparison.

**(subselect)**

Specify a subselect. The subselect must be enclosed by parentheses. For more information about the subselect see Subselect.

**host-variable**

Specify a host-variable. Each host-variable specified must identify a variable that is described in the program under the rules for declaring host-variables. See [Host Variables](#) (see page 520) for more information.

Use a comma to separate the host-variables and enclose the list in parentheses.

**special-register**

Specify a special-register. See [Special Registers](#) (see page 533) for more information on special-registers.

**literal**

Specify a literal. If the expression is numeric, the literal must be numeric.

Use a comma to separate the literals and enclose the list in parentheses.

The following table shows forms of the IN predicate and the predicate form to which it is equivalent.

The IN predicate form:	expression IN expression
is equivalent to:	a basic predicate
of the form:	expression = expression
The IN predicate form:	expression IN (subselect)
is equivalent to:	a quantified predicate
of the form:	expression = ANY (subselect)
The IN predicate form:	where the second operand is a collection of one or more values specified by any combination of literals, host-variables or the keyword USER
is equivalent to:	a quantified predicate
of the form:	expression = ANY (subselect)
except:	the second operand consists of the specified values rather than the values returned by a subselect

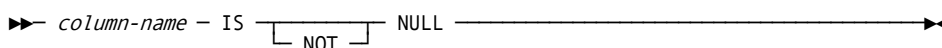
**Example**

The following IN predicate is true only if the value for DEPT is equal to any of the specified literals, A2, B1 or C3. The result is false if the value for DEPT is not equal to any of the literals.

```
DEPT IN ('A2', 'B1', 'C3')
```

## NULL Predicate

Use the NULL predicate to test for null values. Following is the syntax diagram for the NULL predicate:



**Description**

***column-name***

Specify the name of a column in a table or view.

**IS NULL**

Indicates to test the value of the column to determine if it is null.

**NOT**

A keyword. Use NOT to specify that the value of the column be tested to determine if it is not null.

The result of a NULL predicate cannot be unknown. If the value of the column is null, the result is true. If the value is not null, the result is false. If NOT is specified, the result is reversed.

**Example**

The following NULL predicate is true only if the value of PHONENO is null. If the value of PHONENO is not null, the result of the predicate is false.

```
PHONENO IS NULL
```

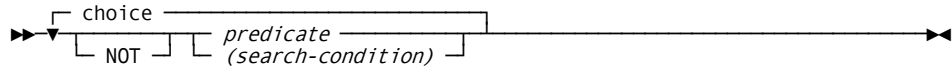


# Chapter 27: Search Conditions

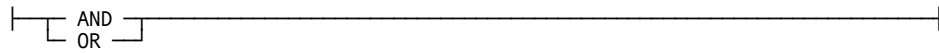
---

A search condition specifies a condition that is "true" or "false" about a given row or group.

Following is the syntax diagram for a search condition:



Expansion of Where choice is as follows



## Description

### ***predicate***

Specify a predicate. For more information about predicates, see [Predicates](#) (see page 581).

### ***(search-condition)***

Specify a search condition. Enclose the search condition within parentheses.

### **AND**

A keyword. This Boolean operator indicates that both conditions joined by this keyword must be satisfied before the result is true.

### **OR**

A keyword. This Boolean operator indicates that only one of the conditions joined by this keyword must be satisfied for the result to be true.

### **NOT**

A keyword. Using NOT negates the result of the predicate or search condition.

The result of a search condition is derived by the application of the specified Boolean operators to the result of each specified predicate. If Boolean operators are not specified, the result of the search condition is the result of the specified predicate.

If the search condition or predicate is preceded by NOT, the result is negated. For example:

- NOT(true) means false
- NOT(false) means true

The following table shows the results when:

1. The value of predicate P is joined with the value of predicate Q by AND.
2. The value of predicate P is joined with the value of predicate Q by OR.
3. Each of the previous operations is preceded by NOT.

P	Q	P AND Q	P OR Q	NOT (P AND Q)	NOT (P OR Q)
T	T	T	T	F	F
T	F	F	T	T	F
F	T	F	T	T	F
F	F	F	F	T	T

Boolean expressions within parentheses are evaluated first. When the order of evaluation is not specified by parentheses:

1. NOT is applied before AND.
2. AND is applied before OR.
3. Operators at the same precedence level are applied from left to right.

### Examples

The following examples show search conditions in WHERE and HAVING clauses. For more information about these clauses, see SELECT.

**Example 1:** This search condition specifies that the value of PNUM must be P3, which is a literal.

```
WHERE PNUM = 'P3'
```

**Example 2:** This search condition specifies that the value of NAME (a string) must be the same as in the host-variable, VAR2.

```
WHERE NAME LIKE :VAR2
```

**Example 3:** This search conditions specifies that the city must be Dallas, and the salary must be greater than \$20,000.

```
WHERE CITY = 'DALLAS' AND SALARY > 20000
```

**Example 4:** This search condition uses an IN predicate and a subselect. The condition specifies that the employee number must be contained in the result table for the SELECT which retrieves only those employee numbers related to department E11.

```
WHERE EMPNO IN (SELECT EMPNO
                FROM EMP
                WHERE DEPTNO = 'E11')
```

**Example 5:** This search condition uses a function and a subselect, which also includes a function. The condition specifies that the maximum salary (for some group) must be less than the average salary for all employees.

```
HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                     FROM EMP)
```



# Chapter 28: SQL Statements

---

This chapter discusses each SQL statement in detail and gives examples.

## Preliminary Information—Lock Levels

In SQL processing, the following statements do not process when the CA Datacom Datadictionary occurrence definition you specify in the statement is protected with a password or Lock Level 1 or 2.

SQL Statement	Occurrence
ALTER	TABLE
DROP	TABLE, VIEW, or SYNONYM

CA Datacom Datadictionary issues a -118 return code and an error message which includes a Datadictionary Service Facility (DSF) return code identifying the problem. See CA Datacom Datadictionary documentation for more information on passwords and lock levels.

## Statements That Support Procedures and Triggers

These statements support procedures and triggers:

- CREATE PROCEDURE (see [CREATE PROCEDURE](#) (see page 620))
- CALL/EXECUTE PROCEDURE —without SET processing support (see [CALL/EXECUTE PROCEDURE](#) (see page 610))
- DROP PROCEDURE (see [DROP](#) (see page 725))
- CREATE TRIGGER/RULE—row level only (see [CREATE TRIGGER/RULE](#) (see page 702))
- DROP TRIGGER/RULE (see [DROP](#) (see page 725))

For an overview and examples of procedures and triggers, see [Procedures and Triggers](#) (see page 70).

## ALTER TABLE

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
ALTER TABLE	YES	YES	YES

**Note:** YES indicates a valid execution method for this statement.

To learn about using SQL keywords in CA Dataquery, see the *CA Dataquery User Guide*.

For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

**Note:** If you are using CA Datacom/DB as part of the CA Datacom/AD environment, you cannot use the ALTER TABLE statement. With regard to table partitioning, ALTER statements may not be issued against a table which is partitioned nor against a partition. An SQL integrity constraint cannot reference a partitioned table, nor a partition of a partitioned table. That is to say, constraints and partitioned tables are mutually exclusive. For more information about table partitioning, see the *CA Datacom/DB Database and System Administration Guide*.

The ALTER TABLE statement is a CA Datacom/DB extension. ALTER TABLE allows you to alter a table definition without having to reload manually. You may change the table's column definition and/or the table's constraints. You may make only one change to a column in an ALTER TABLE statement.

You can query the SYSCONSTRSRC and SYSCONSTRDEP tables (see [Schema Information Tables \(SIT\)](#) (see page 821)) to view information about constraints. You can then use ALTER TABLE to drop the constraint or modify the constraint's selection criteria.

When a table is altered, the prepared statements in plans that reference the table are marked nonexecutable until rebound. An attempt to execute these statements invokes automatic rebind. Depending on the change, the rebind may be successful and require no manual intervention. But if the plans containing referencing statements are currently executing or binding, the ALTER request is aborted. This is always the case when the plan containing the ALTER statement references the table, because this is an attempt to invalidate its own plan and is not allowed.

A Datadictionary Relationship Report can tell you what plans are dependent on a table. See the *CA Datacom Datadictionary Batch Reference Guide* for information on Relationship Reports.

See Results of Using ALTER TABLE for information about the results in CA Datacom Datadictionary of using the ALTER TABLE statement. See [Preliminary Information—Lock Levels](#) (see page 597) for information about lock levels with regard to the ALTER TABLE statement.

The ALTER TABLE statement causes the CONSTRAINT attribute-value to be marked A (for ALTERED) for all TEST and HIST versions of the CA Datacom Datadictionary definition. The next time you try to copy that definition from TEST to PROD status, CA Datacom Datadictionary issues an ALT DSF return code that indicates it cannot be copied because the PROD version has been altered. For details, see the appendix on constrained tables in the *CA Datacom Datadictionary Online Reference Guide* or *CA Datacom Datadictionary Batch Reference Guide*.

The table must be at least half empty before alteration, if using the ALTER TABLE statement causes a table's rows to be reformatted (adding, dropping, or modifying columns). But the table must be *more than* half empty before alteration (that is, less than half full) if the adding, dropping, or modifying of columns causes the size of the table's rows to *grow*.

Issuing an ALTER TABLE statement involves a process that:

1. Copies the contents of the table to the Temporary Table Manager (TTM),
2. Changes the definition of the table, and then
3. Copies the reformatted rows back to the table.

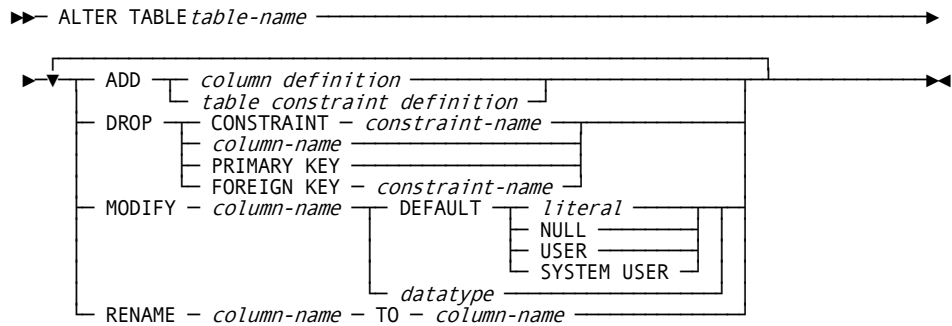
However, a check is made before this operation begins to see if enough space is available in the TTM, and if the answer is no, SQL code -561 (SQLSTATE 57S05) is generated: **TTM TOO SMALL - SEE ERROR ACTION FOR SUGGESTION**. In response to this message you could make the TTM larger, but it can take a long time to perform that operation if there are many rows in the table. Therefore, instead of making the TTM larger consider doing the following:

1. Copy the rows to an external file using the EXTRACT function of DBUTLTY.
2. Convert the rows yourself.
3. For single table areas, do a null LOAD. For multi-table areas, do a REPLACE on the specific table.
4. Perform the ALTER of the empty table.
5. For single table areas, LOAD the converted table. For multi-table areas, do a REPLACE using the converted data.

**Note:** Be aware that because the TTM space check is made at the beginning of the process, there is the possibility that other tasks could use some space and the TTM still become full even if the -561 SQL code was not initially received.

When the NOMAINT option of the CA Datacom/DB Utility (DBUTLTY) ACCESS function is in force, an ALTER TABLE statement receives a CA Datacom/DB return code 94(87), where 87 is a decimal internal return code (hex 57) that tells you no maintenance statements are allowed while NOMAINT is in force.

Following is the syntax diagram for the ALTER TABLE statement:





## Description

### ***table-name***

Specify the name of the table you want to alter. The table-name must identify a table described in the CA Datacom Datadictionary, but not a CA Datacom Datadictionary table.

### **ADD**

Allows you to add column definitions or table constraint definitions.

**Important!** Constraints should not be added to an open table. If a constraint (CHECK, FOREIGN KEY, or PRIMARY KEY) is added to a table which previously had no constraints, the constraint is not enforced in programs which opened the table before the execution of the ALTER TABLE statement. Should this occur, the program should close and reopen the table to activate constraint enforcement for that table. (This caution only applies to record-at-a-time programs referencing tables to which the constraint was added.)

When you use the ALTER TABLE statement to add domain or foreign key constraint(s) to a table, they are added to the Schema Information Tables (SIT) (see [Schema Information Tables \(SIT\)](#) (see page 821)) in check pending status. The CONFIRM function of DBUTLTY is then called to validate existing data against the added constraint(s). Previous existing foreign key(s) are not revalidated. If a violation is found, CA Datacom/DB issues an SQL warning message and changes the table's Directory (CXX) status to prevent access to the table's rows.

**Note:** You must CONFIRM tables in the following order, from referenced-to-referencing, or parent-to-child, that is to say, the referenced table cannot be in check pending status. For example, if there is a foreign key from a REMARKS table to a LINE\_ITEM table on PO\_Nbr and Line\_Item\_Nbr, and the LINE\_ITEM table has a foreign key that references the PO table on PO\_Nbr, the PO table must be loaded first, then the LINE\_ITEM table can be confirmed, then the REMARKS table can be confirmed. Therefore, the confirm order would be: from PO table to LINE\_ITEM table to REMARKS table.

If violations are found, with the table in check pending status you can do one of two things:

1. You can execute the CONFIRM function of DBUTLTY to delete the row(s) in error and optionally write them to an exception table for further action. To identify the violated constraint(s), after writing the row(s) in error to an exception table you can write a program to read those row(s) and attempt to insert them back into the original table so that the violated constraint(s) are reported.
2. Or you can use the ALTER TABLE statement to drop the violated constraint(s), causing the table to be reconfirmed and taken out of check pending status. If all the added constraints are dropped, the CONFIRM function of DBUTLTY notes that there are no constraints in check pending status in the SIT and resets the table's Directory (CXX) status to allow access to the table's rows.

If you try to add a new constraint other than domain or foreign key constraint(s) previously described and there is data already in the table that violates the constraint, CA Datacom/DB issues an error message. You can correct the error by either changing the constraint definition or removing the data from the table.

### ***column definition***

When a new column is added (through a column definition), rows that already exist in the table receive NULL values in the added column unless NOT NULL WITH DEFAULT is specified in the column constraint definition. If NOT NULL WITH DEFAULT is specified, they receive zero (0) for NUMERIC data types, blanks for fixed-length strings, 1/1/1 for dates, 0:00:00 for times, and 1/1/1-0:00:00.000000 for timestamps.

The new column is added at the end of the row.

When a table is created through SQL, an element with the name SQLEL is added spanning the full row. If the SQLEL element exists, the length of this new column is added. If the SQLEL element does not exist, it is added.

See [Column Definition](#) (see page 683) for information about the column definition and [Column Constraint Definition](#) (see page 684) for information about the column constraint definition.

**table constraint definition**

See [Table Constraint Definition](#) (see page 687).

**DROP**

Allows you to drop CONSTRAINTs, columns, PRIMARY KEYs, or FOREIGN KEYs. If you try to drop a constraint by name during an ALTER TABLE statement and the specified constraint name is not found, CA Datacom/DB issues an error message. You can query the SYSCONSTRSRC and SYSCONSTRDEP tables (see Schema Information Tables (SIT)) to determine the correct name of the constraint.

You cannot drop a primary or unique constraint if there is one or more foreign references to it. In such cases, CA Datacom/DB issues an error message. You can query the SYSCONSTRSRC and SYSCONSTRDEP tables (see [Schema Information Tables \(SIT\)](#) (see page 821)) to locate the foreign reference. You can then remove all foreign references before dropping the primary or unique constraint.

**Note:** When a PRIMARY KEY or UNIQUE constraint is dropped, the corresponding index (key) remains but is made nonunique, even if the key was originally created using CA Datacom Datadictionary.

You cannot drop a WITH CHECK OPTION constraint from a view definition by using the ALTER TABLE statement. To drop a WITH CHECK OPTION constraint from a view definition, use the DROP statement to drop the view definition, then use the CREATE VIEW statement to re-create the view definition without the WITH CHECK OPTION constraint.

**Note:** Dropping the view and re-creating it requires the rebinding of all dependent statements and views. You also have to re-create any views which were dependent on the dropped view.

A column may not be dropped if it is:

- The only column in any element,
- The only column of a key, or
- Involved in any constraint, that is to say, a FOREIGN KEY, PRIMARY KEY, UNIQUE, or CHECK constraint. If you attempt to drop a column involved in any constraint, you receive an SQL -242 return code (CONFLICT ALTERING COLUMN).

DROP does not process and you receive a -118 return code when the CA Datacom Datadictionary entity-occurrence definition of the table or view you specify is protected with a password or a Lock Level 1 or 2. The error message also includes a Datadictionary Service Facility (DSF) return code. The DSF return codes are:

- IPW (for password protected)
- IOR (for Lock Level 1 protected)
- NTF (for Lock Level 2 protected)

See [Deleting SQL Objects](#) (see page 431) for more information, and see the CA Datacom Datadictionary documentation for information on passwords and lock levels.

**CONSTRAINT *constraint-name***

Specifies the CONSTRAINT to be dropped. See [Column Constraint Definition](#) (see page 684) for information about the CONSTRAINT *constraint-name*.

***column-name***

Specifies the name of a column. The column must belong to the table you have specified with the table-name.

**PRIMARY KEY**

Specifies that the PRIMARY KEY constraint for a table name is to be dropped. See [Column Constraint Definition](#) (see page 684) for information about the PRIMARY KEY.

**FOREIGN KEY *constraint-name***

Specifies the FOREIGN KEY to be dropped. See [Referential Constraint Definition](#) (see page 689) for information about the FOREIGN KEY.

**MODIFY**

Allows you to modify a column's DEFAULT or data type.

You cannot modify a column that is involved in any constraint, that is to say, a FOREIGN KEY, PRIMARY KEY, UNIQUE, or CHECK constraint. If you attempt to modify a column that is involved in any constraint, you receive an SQL -242 return code (CONFLICT ALTERING COLUMN).

**DEFAULT**

Allows you to specify a default.

**literal**

Specifies a literal as the default. The literal you specify must be consistent with the data type of the specified column.

A user-supplied DEFAULT literal can be up to 20 bytes long, or the length of the column involved, whichever is shorter. A default value may be specified for a character column where the column is greater than 20 bytes long, but the default literal itself is limited to 20 bytes, with the remaining bytes padded with blanks by the system.

**NULL**

Specifies a NULL value as the default.

**USER**

Specifies the current authorization ID as the default.

**SYSTEM USER**

This CA Datacom/DB extension specifies the accessor ID of the currently signed-on user as the default.

**datatype**

A column's data type may be changed with the following rules:

- DATE, TIME, TIMESTAMP: these data types may not be modified.
- Changing the length of a character field results in padding with blanks or truncation. You receive an error message if truncation would delete non-blank data.

**Important!** All data types that are known to CA Datacom Datadictionary as special data types but are unknown to SQL are treated by SQL as character. Therefore, if you alter one of these columns, the ALTER uses the character rules.

- Any of the numeric data types may be converted to any other numeric data type (FLOAT, REAL, DOUBLE PRECISION, DECIMAL, SMALLINT, INTEGER, NUMERIC). Conversion of values from the old to the new data type is treated with the same truncation rules as assignment. (See [Basic Operations \(Assignment and Comparison\)](#) (see page 501).)

See [Data Types](#) (see page 695) for more information about data types.

**RENAME *column-name* TO *column-name***

Allows you to rename a column. This changes the SQL name of the column throughout the CA Datacom Datadictionary and SIT tables. It does not correct any external source statements.

You cannot rename a column that is involved in any constraint, that is to say, a FOREIGN KEY, PRIMARY KEY, UNIQUE, or CHECK constraint. If you attempt to rename a column that is involved in any constraint, you receive an SQL -242 return code (CONFLICT ALTERING COLUMN).

## Processing

In one ALTER TABLE statement, all ADD column-name, DROP column-name, and MODIFY column-name with data type functions are done first and as one group. Processing this group is done in the following sequence:

1. Lock the table.
2. Read the first/next row in the table.
3. Add the row to the SQL Temporary Table (TTM).
4. Delete the row.
5. Loop back through all the rows.
6. Update the definitions in the CA Datacom Datadictionary and Directory (CXX).
7. Read the records from the SQL Temporary Table (TTM).
8. Reformat the row by adding new columns to the end, deleting any dropped columns, and changing the format and/or length of any modified column.
9. Add the row to the original table.
10. Loop back to process all rows.

Primary and unique constraints are validated as rows are added back in step 9. If a duplicate is found, the entire ALTER statement is backed out.

Foreign keys and domain constraints are added after all rows have been added back. After adding the constraint definition(s), the rows of the table are read in Native Key sequence to validate the constraints. If a row violates one or more domain constraints, the table is placed in check pending status and an SQL return code 170 is returned. You must use the CONFIRM function of DBUTLTY to remove the check pending status before the table can be used. If a row violates foreign keys, the ALTER TABLE statement fails with an SQL return code -176.

If the ALTER TABLE statement cannot complete, it is rolled back to its beginning. If the ALTER TABLE process is force checkpointed (because of the size of the Log Area (LXX) and the concurrent activity), the rollback is incomplete. Therefore, before altering large tables, ensure that the Log Area (LXX) is large enough and take a backup of the data area.

### Example 1

Modify an existing table column decimal number to precision 6 with a scale of 2 decimals. This assumes the previous column was such that no data is lost.

1. Table name: DEPTTBL
2. Column name: PROJSTAFF
3. Data type: DECIMAL

```
EXEC SQL
      ALTER TABLE DEPTTBL
      MODIFY PROJSTAFF DECIMAL(6,2)
END-EXEC
```

### Example 2

Modify two columns to allow for more character data than previously defined.

1. Table name: DEPTTBL
2. Column names: ADMDEPT (to be 3 characters) and DEPTNAME (to be 30 characters)

```
EXEC SQL
      ALTER TABLE DEPTTBL
      MODIFY ADMDEPT CHAR(3)
      MODIFY DEPTNAME CHAR(30)
END-EXEC
```

### Example 3

Alter table ORDERS to prevent the acceptance of any order that would require the financing of a purchase of less than \$1000.00.

1. Require a down payment of 100 percent on orders of less than \$1000.00.
2. Give a meaningful constraint name: ORDERS\_MIN\_AMT\_FINANCED.

```
EXEC SQL
      ALTER TABLE ORDERS
      ADD CHECK (GROSS_AMOUNT >= 1000.0 OR PERCENT_DOWN_PMT = 100.0)
      CONSTRAINT ORDERS_MIN_AMT_FINANCED
END-EXEC
```

### Example 4

Respond to customers' requests to allow financing on small purchases.

1. Table name: ORDERS
2. Remove constraint: ORDERS\_MIN\_AMT\_FINANCED  
EXEC SQL  
    ALTER TABLE ORDERS  
        DROP CONSTRAINT ORDERS\_MIN\_AMT\_FINANCED  
END-EXEC

### Example 5

Prevent the acceptance of orders from customers who do not pay their bills by ensuring that any order accepted comes from a customer whose name is on a "customers in good standing" list.

1. Table containing customers in good standing: PAYING\_CUSTOMERS
2. Table to be protected: ORDERS
3. Columns on which a match is required:  
    ORDERS.CUSTOMER\_NAME to PAYING\_CUSTOMERS.COMPANY\_NAME
4. Constraint name: ORDERS\_TO\_PAYING\_CUST\_NAME  
EXEC SQL  
    ALTER TABLE ORDERS ADD FOREIGN KEY  
        (CUSTOMER\_NAME) REFERENCES PAYING\_CUSTOMERS (COMPANY\_NAME)  
        CONSTRAINT ORDERS\_TO\_PAYING\_CUST\_NAME  
END-EXEC



## Example 6

Prevent the entrance of a shipment into the system unless there is an outstanding order for the goods to be shipped.

1. Table containing outstanding orders: ORDERS
2. Table to be protected: CURRENT\_SHIPMENTS
3. Columns requiring a match:  
CURRENT\_SHIPMENTS columns ORDER\_ID and SHIPMENT\_NUM to ORDERS  
columns ORDER\_ID and SHIPMENT\_ID
4. Constraint name: SHIPMENTS\_TO\_ORDERS\_IDS\_MATCH  
EXEC SQL  
ALTER TABLE CURRENT\_SHIPMENTS ADD FOREIGN KEY  
(ORDER\_ID, SHIPMENT\_NUM) REFERENCES  
ORDERS(ORDER\_ID, SHIPMENT\_ID)  
END-EXEC

**Note:** The columns referenced by a foreign key must correspond to the primary key of the referenced table.

## Example 7

Remove the foreign key restriction from table CURRENT\_SHIPMENTS in order to ship goods that were mistakenly left out of the original shipments, when the order is no longer current.

- Foreign key name: SHIPMENTS\_TO\_ORDERS\_IDS\_MATCH  
EXEC SQL  
ALTER TABLE CURRENT\_SHIPMENTS  
DROP FOREIGN KEY SHIPMENTS\_TO\_ORDERS\_IDS\_MATCH  
END-EXEC

**Note:** If a domain constraint is added to a table containing rows that violate the constraint, you receive an SQL return code to warn you of this fact. The constraint is added, but the table is placed in "check" status to make it unusable until the problem has been corrected by either running the CONFIRM function of DBUTLTY with the DELETE=YES option specified to delete violating rows, or by dropping the constraint and then running CONFIRM.

A similar situation occurs when adding a foreign key for which the referenced table lacks rows to match all of the values in the referencing table. The same options (previously described) are available to correct the problem. However, in the case of dropping a foreign key to correct the situation, an automatic CONFIRM is initiated for you so that you do not need to perform this step yourself.

## Assignment Statement

For details about this statement, see [Assignment Statement](#) (see page 638).

**Note:** This statement is executed from within the Compound Statement (see [Compound Statement](#) (see page 642)).

## CALL/EXECUTE PROCEDURE

For an overview and examples of procedures and triggers, see [Procedures and Triggers](#) (see page 70).

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
CALL EXECUTE PROCEDURE	YES if no parms passed	YES	YES if no parms passed

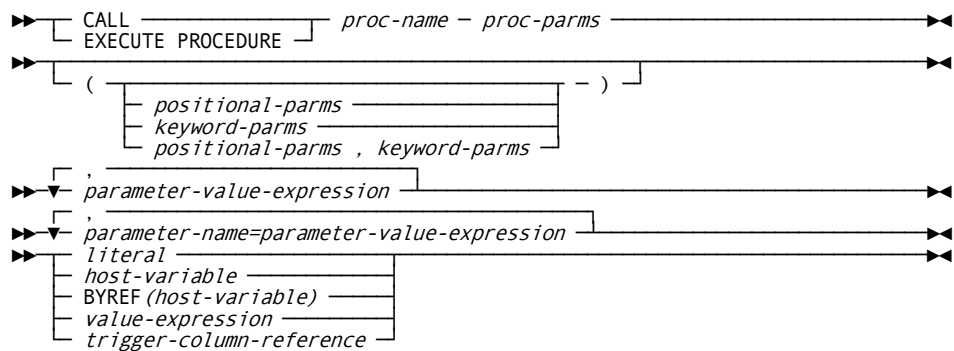
**Note:** YES indicates a valid execution method for this statement.

A CALL or EXECUTE PROCEDURE statement is used to call user-written procedure logic. This statement can be coded directly by the end-user or can be embedded in a trigger definition. Please note that procedures themselves can contain CALL/EXECUTE PROCEDURE statements, subject to the following discussion of nesting limitation.

A procedure can also cause procedures to execute (*trigger* them) by executing INSERT, UPDATE, or DELETE statements. The depth of nesting of these recursive procedure calls is limited by the PROCEDURE Multi-User startup option that can be used to limit or eliminate nesting.

If null indication variables are passed to the procedure, a list of pointers to the indicators follows the list of pointers to the variables themselves. Although the pointers are contiguous, the indicators themselves are not.

**Note:** Because SQL has no way of verifying the compatibility of the user-written code with the procedure defined by the CALL/EXECUTE PROCEDURE statement, it is the sole responsibility of the creator of the procedure to ensure that the CALL/EXECUTE 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.



## CALL

(Required)

## EXECUTE PROCEDURE

(Required)

### *proc-name*

Specify a procedure name. The procedure name cannot be specified as a host variable reference.

ANSI supports *overloading* of the procedure name, with duplicate procedure names resolved to procedure definitions using the layout of the parameter list. For syntax compatibility purposes, a second, *specific* name may be given to a procedure to uniquely identify it, but it must match the nonspecific name, and the nonspecific name must be unique.

### *parameter-name*

Enter the name of the parameter as specified in the CREATE PROCEDURE statement.

### *literal*

Specify a *literal* (value) for the parameter.

### *host-variable*

Specify a *host-variable* by naming a variable that is described in the program in accordance with the rules for declaring host variables.

**BYREF(*host-variable*)**

This clause, supported for compatibility purposes, is treated as a host variable reference.

***value-expression***

Specify a value expression (value expressions are used in procedure parameter lists).

***trigger-column-reference***

(Valid only in a CREATE TRIGGER or CREATE RULE statement.) A reference to a base table column value through a correlation name defined in the old-row/new-row syntax of the CREATE TRIGGER/CREATE RULE statements.

## CASE Statement

For details about this statement, see [CASE Statement](#) (see page 640).

**Note:** This statement is executed from within the Compound Statement (see [Compound Statement](#) (see page 642)).

## CLOSE

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
CLOSE		YES	

**Note:** YES indicates a valid execution method for this statement. For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

The CLOSE statement closes a cursor. If a temporary table is created when the cursor is opened, the table is destroyed when the cursor is closed.

Following is the syntax diagram for the CLOSE statement:



## Description

### ***cursor-name***

The name of a cursor that is defined in a DECLARE CURSOR statement in your program. The DECLARE CURSOR statement must precede the CLOSE statement in your source program. When the CLOSE statement is executed, the cursor must be in the open state.

## Processing

If your program terminates without closing an open cursor, the cursor is closed by CA Datacom/DB.

## Example

See DECLARE CURSOR's "Example 1" in [Example1](#) (see page 715).

## COMMENT ON

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
COMMENT ON	YES	YES	YES

**Note:** YES indicates a valid execution method for this statement. To learn about using SQL keywords in CA Dataquery, see the *CA Dataquery User Guide*. For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

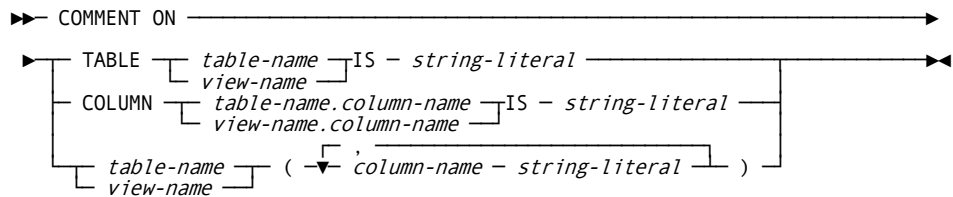
The COMMENT ON statement is a CA Datacom/DB extension. COMMENT ON adds or replaces comments in CA Datacom Datadictionary. The comments apply to tables, views or columns.

If a comment does not exist for the table, view or column, the comment you specify is added to text in CA Datacom Datadictionary. If a comment already exists for the table, view or column, and you specify a new comment, the existing comment is replaced by the new comment.

You cannot retrieve a comment through SQL. You can display the text created by the COMMENT ON statement through the CA Datacom Datadictionary batch and online functions. See the *CA Datacom Datadictionary User Guide* for information about displaying text.

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. See the *CA Datacom Datadictionary User Guide* for more information about specifying additional text.

Following is the syntax diagram for the COMMENT ON statement:



## Description

### TABLE

Indicates you want to comment on a table or view.

#### **table-name or view-name**

Must identify a table or view described in the CA Datacom Datadictionary. The comment is placed in text in CA Datacom Datadictionary.

### COLUMN

Indicates you want to comment on a column.

#### **table-name.column-name or view-name.column-name**

The name of the column, qualified by the name of the table or view in which it appears. The column must be described in the CA Datacom Datadictionary. The comment is placed in text in CA Datacom Datadictionary.

**IS string-literal**

The string-literal can be any character string literal enclosed between apostrophes.

**column-name IS string-literal**

To comment on more than one column in a table or view, do not use TABLE or COLUMN. Give the table or view name and a list of the form:

(column-name IS 'string-literal', column-name IS 'string-literal', column-name IS 'string-literal')

The column comments must be separated by commas and the list must be enclosed with parentheses. All columns named must appear in the same table or view and the table or view must be described in the CA Datacom Datadictionary.

## Example 1

Enter a comment on table EMP.

```
EXEC SQL
  COMMENT ON TABLE EMP
  IS 'REFLECTS 1ST QTR 87 REORGANIZATION'
END-EXEC
```

## Example 2

Enter a comment on view MGR.

```
EXEC SQL
  COMMENT ON VIEW MGR
  IS 'VIEW OF TABLE EMP THAT ARE MANAGERS'
END-EXEC
```

## Example 3

Enter a comment on the EMPNBR column of table EMP.

```
EXEC SQL
  COMMENT ON COLUMN EMP.EMPENBR
  IS 'EMPENBR IS UNIQUE'
END-EXEC
```

## Example 4

Enter comments on two columns in table DEPTTBL.

```
EXEC SQL
  COMMENT ON DEPTTBL
  (MGRNBR IS 'EMPLOYEE NUMBER OF DEPARTMENT MANAGER',
  ADMDEPT IS 'DEPARTMENT NUMBER OF ADMINISTERING DEPARTMENT')
END-EXEC
```

## COMMIT WORK

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
COMMIT WORK	YES	YES	

**Note:** YES indicates a valid execution method for this statement. For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

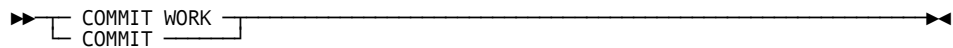
The COMMIT WORK statement terminates a unit of recovery and commits the database changes that were made by that unit of recovery. The commit closes all open cursors.

**Note:** If a cursor is defined WITH HOLD, it stays open when a COMMIT WORK is executed. Any record-at-a-time command that commits the logical unit of work (for example LOGCP, LOGCR) works the same way. See the description of the WITH HOLD clause in DECLARE CURSOR.

A commit point is the moment in the sequence of operations at which the commit is actually done.

Following is the syntax diagram for the COMMIT WORK statement:

**Note:** COMMIT is a CA Datacom/DB extension.



## Description

### COMMIT WORK

The COMMIT WORK statement is the commit operation. The unit of recovery in which the statement is executed is terminated and a new unit of recovery is initiated. All changes made by CREATE, COMMENT ON, DROP, INSERT, UPDATE and DELETE statements executed during the unit of recovery are committed. A COMMIT WORK releases all locks not acquired by a LOCK TABLE statement. See [LOCK TABLE](#) (see page 751) for an explanation of the duration of explicitly acquired locks.

### COMMIT

This CA Datacom/DB extension has the same effect as COMMIT WORK.



A unit of work is made up of 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.

A unit of recovery is a sequence of operations within a unit of work. A unit of recovery is initiated by:

- The initiation of a unit of work.
- The termination of a previous unit of recovery.

A unit of recovery is terminated by:

- A commit operation.
- A rollback operation.
- The termination of a unit of work.

A commit or rollback operation affects only the results of SQL statements executed within a single unit of recovery.

Uncommitted database changes made in a unit of recovery may or may not be perceived by other units of work depending on the isolation level selected.

Uncommitted database changes made in a unit of recovery can be backed out by CA Datacom/DB.

Committed database changes can be perceived by other units of recovery and cannot be backed out by CA Datacom/DB.

CA Datacom/DB database changes are also committed when a unit of recovery terminates normally.

## Example

The following statement commits alterations to the database made since the last commit point.

```
EXEC SQL
      COMMIT WORK
END-EXEC
```

# CREATE INDEX

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
CREATE INDEX	YES	YES	YES

**Note:** YES indicates a valid execution method for this statement. To learn about using SQL keywords in CA Dataquery, see the *CA Dataquery User Guide*. For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

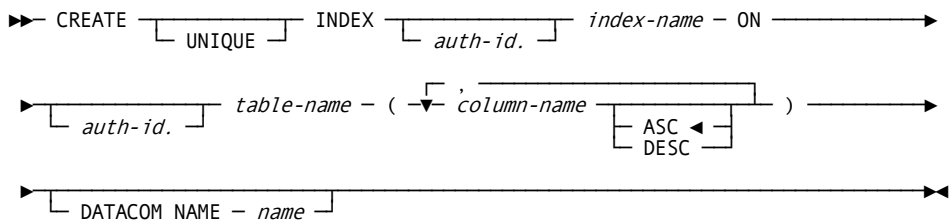
The CREATE INDEX statement is a CA Datacom/DB extension. CREATE INDEX defines a new index on one or more columns of a base table. Adding an index to a table may improve the performance of some queries which reference that table.

When a CREATE INDEX statement successfully executes, a KEY entity-occurrence is defined in CA Datacom Datadictionary in PROD status. The five-character key name is generated for you by CA Datacom Datadictionary. The key's attributes include MASTER-KEY=N, NATIVE-KEY=N, INCLUDE-NIL-KEY=Y, and UNIQUE=N. For information about specifying SQL key selection override keys in either the correlation name or synonym name of a query, see *Overriding SQL Key Selection*.

CREATE INDEX causes all plans dependent on the indexed table to be marked invalid. You can run a Datadictionary Relationship Report to find out what plans are dependent on a table. See the *CA Datacom Datadictionary Batch Reference Guide* for information on Relationship Reports.

**Note:** When the NOMAINT option of the CA Datacom/DB Utility (DBUTLTY) ACCESS function is in force, a CREATE INDEX statement receives a CA Datacom/DB return code 94(87), where 87 is a decimal internal return code (hex 57) that tells you no maintenance statements are allowed while NOMAINT is in force. With regard to table partitioning, CREATE statements may not be issued against a table which is partitioned nor against a partition. For more information about table partitioning, see the *CA Datacom/DB Database and System Administration Guide*.

Following is the syntax diagram for the CREATE INDEX statement:



---

## Description

### **UNIQUE**

If UNIQUE is specified, the created index enforces value uniqueness and provides an efficient means of both retrieving rows containing specific data values and of ordering the data. The index is rejected if the table contains non-unique values.

### ***auth-id.***

An optional identifier of the schema for the index and table. If specified for the index-name, it must be the same as the authorization ID of the table you have specified with the table-name. Use a period (.) to concatenate the authorization ID to the index name or table name (for example, *auth-id.table-name*).

### ***index-name ON***

The name for the index you are creating. The index-name you supply, including the implicit or explicit qualifier (authid), must not identify an index already described in CA Datacom Datadictionary.

### ***table-name***

Specify the name of the table to be indexed. Must be a table described in CA Datacom Datadictionary.

### ***(column-name)***

Specifies the name of a column or a list of columns. The column(s) must belong to the table you have specified with the table-name. If more than one column name is listed, each name must be different and separated by commas. The name of a column or a list of columns must be enclosed in parentheses.

### **ASC**

Places the values of the column in ascending order. ASC is the default order.

### **DESC**

Places the values of the column in descending order.

### **DATACOM NAME *name***

Specifies a five-character DATACOM NAME name (a KEY entity-occurrence in CA Datacom Datadictionary ) for the index. If you do not specify a DATACOM NAME name, CA Datacom Datadictionary generates a name for you. (This is not the CA Datacom Datadictionary entity-occurrence name.)

## Processing

CREATE INDEX processing is done by:

1. Locking the table to be indexed,
2. Copying the rows to a temporary table while deleting them from the original table,
3. Adding the definition of the new index, and then
4. Adding the rows from the temporary table back to the original table.

Because of the amount of processing involved, adding an index through the CREATE INDEX statement may not be appropriate for tables with a large number of rows. You may wish to use DBUTLTY's BACKUP function to unload the table's area and then use DBUTLTY's LOAD function to null load the area before executing the CREATE INDEX statement. After adding the new key definition by executing the CREATE INDEX statement, DBUTLTY's LOAD function can be used to reload the area.

**Note:** For information on using DBUTLTY, see the *CA Datacom/DB DBUTLTY Reference Guide*.

The sum of the lengths of all columns to be included in the index must be no more than 180.

## Example

The following example creates an EMPLOYEE\_INDEX index on table-name EMPLOYEES, column-name EMPNO.

```
EXEC SQL
    CREATE INDEX EMPLOYEE_INDEX ON EMPLOYEES (EMPNO)
END-EXEC
```

## CREATE PROCEDURE

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
CREATE PROCEDURE	YES/NO	YES	YES/NO

**Note:** YES indicates a valid execution method for this statement.

YES/NO means YES for External Procedures but NO for SQL Procedures (see following explanations of [External Procedures](#) (see page 622) and [SQL Procedures](#) (see page 622)).

External Procedure definitions and SQL Procedures are created with this statement. External Procedures are written in a host programming language (COBOL, PL/I, C, or ASSEMBLER) with SQL statements imbedded in their programs. For an overview and examples of External Procedures and triggers, see [Procedures and Triggers](#) (see page 70). SQL Procedures are written in SQL and have all of their execution-time logic included in an SQL CREATE PROCEDURE statement.

There are three ways to execute procedures:

- CALL statements (see [CALL/EXECUTE PROCEDURE](#) (see page 610))
- EXECUTE PROCEDURE statements (see [CALL/EXECUTE PROCEDURE](#) (see page 610))
- Triggers (see [CREATE TRIGGER/RULE](#) (see page 702))

We recommend using PLNCLOSE=T for procedures instead of PLNCLOSE=R. For External Procedures, the PLNCLOSE= specification is determined by the Preprocessor options used for the procedure body compilation. For SQL Procedures, the PLNCLOSE= specification is determined by the options used by the plan in which the CREATE PROCEDURE statement is executed. When using DBSQLPR, specify PLNCLOSE=T or allow PLNCLOSE to default to that value.

When you code procedures that perform INSERTs or UPDATEs, keep in mind that if they are called or triggered (directly or indirectly) from an application that reads the same table on which that procedure is operating, the procedure affects the results of the calling or triggering application and, in extreme cases, can cause loops to occur. For example, if an application is performing a searched UPDATE of a table, and the UPDATE triggers a procedure that does UPDATEs itself, the procedure could cause an endless loop by moving previously updated rows back into the traversal path of the controlling searched update. Because CA Datacom/DB cannot prevent such a situation from occurring, it is important that you make every effort to ensure your coding of a procedure does not cause this to occur.

Triggers whose procedures perform INSERTs, UPDATEs, or DELETEs have the potential to trigger themselves recursively, triggering themselves immediately, or passing through other triggers first. However, be aware that the *maximum recursion depth* specified in the PROCEDURE Multi-User startup option cannot be exceeded.

## External Procedures

External Procedure logic is created by executing a preprocessor against user-written host-language source code. The program source code for the procedure must be preprocessed before the CREATE PROCEDURE statement is allowed to run.

The following in the CREATE PROCEDURE statement distinguishes External Procedures from SQL Procedures:

- External Procedures allow any valid choice for the language in the *proc-attributes* syntax except SQL.
- External Procedures allow, in the *proc-body* syntax, the use of the *proc-external* and *parameter-style* clauses, but not the *proc-SQL-stmt* clause.

In a z/OS environment, COBOL, PL/I, and C procedures must be made Language Environment (LE) conforming by being written and compiled using COBOL for z/OS, PL/I for MVS, and z/OS C/C++. Assembler procedures must also be made LE-conforming by use of the CEEENTRY and associated macros.

In a z/VSE environment, COBOL, PL/I, and C procedures must be made Language Environment (LE) conforming by being written and compiled using COBOL for z/VSE, PL/I for z/VSE, IBM C for z/VSE, and High Level Assembler.

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 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.

**Note:** As described in Multi-User Facility Considerations for Procedures, you should add to the Multi-User Facility library concatenation the libraries containing the programs to be executed as procedures, along with any associated subroutines. Also, if an existing procedure is being replaced, do a NEWCOPY console command, as described in the *CA Datacom/DB Database and System Administration Guide*, to that procedure so that the next job executes the latest copy. For additional information, see Multi-User Facility Considerations for Procedures.

## SQL Procedures

SQL procedures support a basic procedural language with support for the following:

- declaration of SQL variables
- value assignment
- decision-making logic (IF-THEN statement) and CASE statement
- looping constructs (LOOP, REPEAT-UNTIL, and WHILE statements)

- error and condition handling, including user-defined conditions (see [Diagnostics and Condition Handling](#) (see page 634))
- As many SQL statements as the size of your RWTS area allows

For SQL Procedures, the CREATE PROCEDURE statement is specified as follows:

- SQL is specified for LANGUAGE in the *language* syntax.
- In the *proc-body* syntax, a *proc-SQL-stmt* clause is specified. The *proc-SQL-stmt* clause is composed of SQL Procedure statements that are coded and executed directly. For instructions about coding a list of statements and program-like logic, see the compound statement information in [Compound Statement](#) (see page 642).

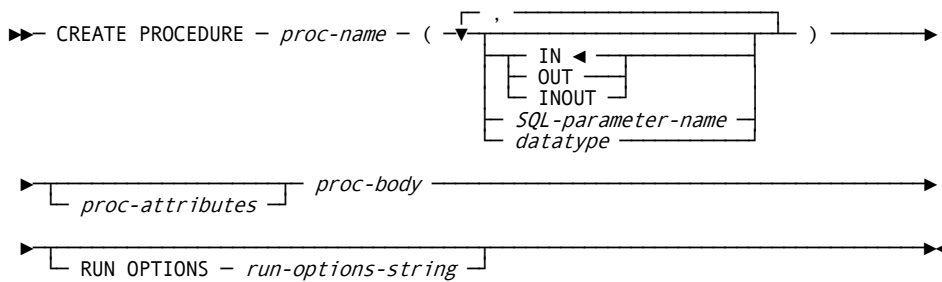
Following is a list of the SQL statements that support SQL Procedures:

- [Assignment Statement](#) (see page 638)
- [CASE Statement](#) (see page 640) (different from a case expression)
- [Compound Statement](#) (see page 642) (the building block from which SQL Procedures are mainly composed)
- condition declaration
- [DATACOM DUMP Statement](#) (see page 647)
- [GET DIAGNOSTICS Statement](#) (see page 649)
- [IF-THEN Statement](#) (see page 652)
- [ITERATE Statement](#) (see page 653)
- [LEAVE Statement](#) (see page 654)
- SQL variable declaration
- looping constructs
  - [LOOP Statement](#) (see page 655)
  - [REPEAT-UNTIL Statement](#) (see page 658)
  - [WHILE Statement](#) (see page 675)
- [RAISE ERROR Statement](#) (see page 657)
- [RESIGNAL Statement](#) (see page 660)
- [SIGNAL Statement](#) (see page 663)
- [SIMULATE DATACOM PROCEDURE Statement](#) (see page 665) (DBSQLPR only)

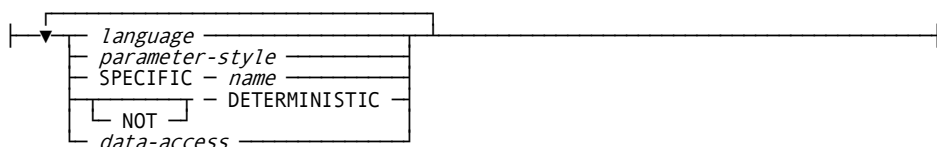
## CREATE PROCEDURE Syntax and Description

Following is the syntax for the CREATE PROCEDURE statement:

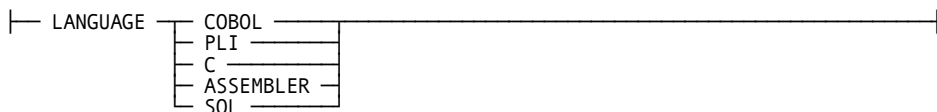
**Note:** SQL-parameter-name is required for SQL Procedures and, though optional for all others, is recommended for all others as well. See the separate *proc-body* syntax diagram that follows.



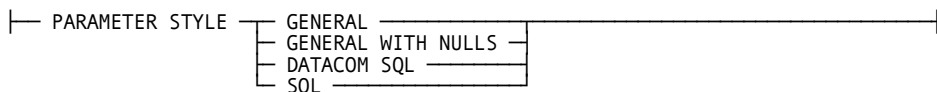
Expansion of Where *proc-attributes* is defined as



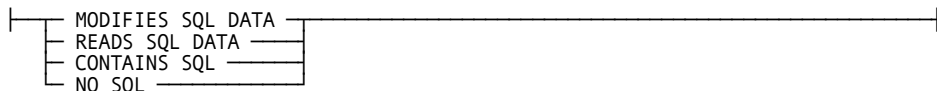
Expansion of Where *language* is defined as



Expansion of Where *parameter-style* is defined as



Expansion of Where *data-access* is defined as



**Note:** All of the *proc-attributes* choices can be used together, but each can only be used once. If you are using SQL Procedures, *data-access* and *parameter-style* do not apply.

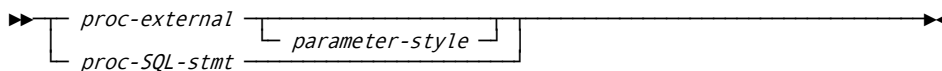


You can specify the parameter style only once, either here as part of the procedure attribute syntax or as part of the EXTERNAL syntax. Parameter style applies only to External Procedures. It does not apply to SQL Procedures.

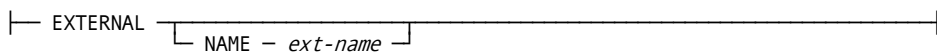
**Note:** The *proc-external* clause is only used for External Procedures.

The optional *parameter-style* clause is only used for External Procedures.

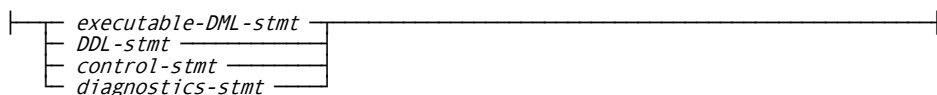
The *proc-SQL-stmt* clause is only used for SQL Procedures.



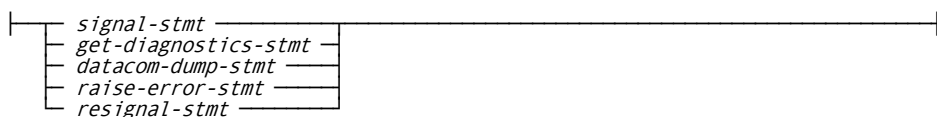
Expansion of Where *proc\_external* is defined as



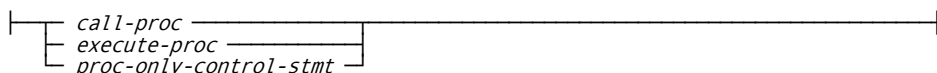
Expansion of Where *proc-SQL-stmt* is defined as



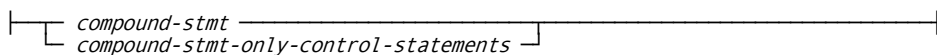
Expansion of Where *diagnostics-stmt* is defined as



Expansion of Where *control-stmt* is defined as



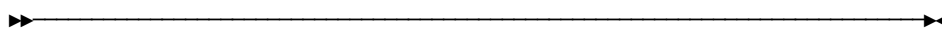
Expansion of Where *proc-only-control-stmt* is defined as



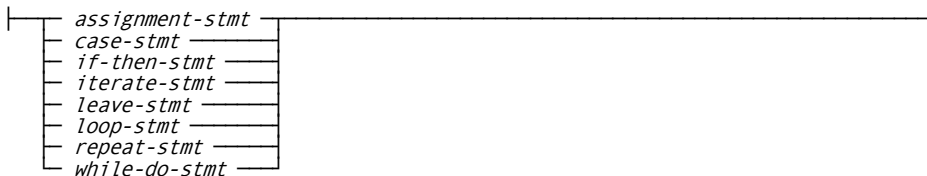
**Note:** Syntax for the compound-stmt-only-control-statement is given in a separate diagram on the next page.

**Note:** The executable-DML-stmt is any DML statement except for DECLARE CURSOR (DECLARE CURSOR is supported as part of the compound statement, see [Compound Statement](#) (see page 642)). For a list of DML statements, see SQL Statements.

See the separate *diagnostics-stmt* syntax diagram that follows and see the GET DIAGNOSTICS statement on [GET DIAGNOSTICS Statement](#) (see page 649).



Expansion of Where compound-stmt-only-control-statements are defined as



Following is the description of the CREATE PROCEDURE syntax:

**proc-name**

(Required) For the *proc-name* specify the SQL-name of the procedure name. The name can be qualified with an authorization ID. This name is an SQL-identifier.

We recommend that you use unique procedure names with regard to the names of any record, table, synonym, view, or constraint. If you code and reference a procedure parameter, for example, whose name is identical to a column name that is also referenced inside a particular SQL Procedure, uniqueness in naming would allow you to use the procedure name and table name to distinguish a procedure parameter reference from a reference to the database-table column.

ANSI supports *overloading* of the procedure name, with duplicate procedure names resolved to procedure definitions using the layout of the parameter list. For syntax compatibility purposes, a second, *specific* name may be given to a procedure to uniquely identify it, but it must match the nonspecific name, and the nonspecific name must be unique.

**IN/OUT/INOUT**

(Optional) Part of the SQL parameter definition. Specifies to SQL whether this parameter is used for input (IN), output (OUT), or both (INOUT). If this parameter is not specified, the default (IN) is used.

**Valid Entries:**

IN, OUT, INOUT

**Default Value:**

IN

**SQL-parameter-name**

*(Optional)* The *SQL-parameter-name* is required for SQL Procedures and, though optional for all others, is recommended for all others as well. The *SQL-parameter-name* is the name of a parameter passed to a SQL Procedure (a LANGUAGE SQL procedure). When used in a SQL Procedure containing an SQL variable with a conflicting (matching) name, or a table or view reference where the table or view contains a conflicting column name, the name should be qualified by using the procedure name.

**datatype**

Part of the SQL parameter definition. Specify a datatype.

**Valid Entries:****Default Value:**

(No default)

**LANGUAGE COBOL/PLI/C/ASSEMBLER/SQL**

Part of the procedure attributes definition.

Specify COBOL, PLI, C, or ASSEMBLER as the programming language when writing a External Procedure.

Specify SQL when writing a SQL Procedure. When you create a LANGUAGE SQL procedure, CA Datacom/DB creates a plan to hold the SQL statements contained by the procedure. The plan name that is created consists of the first 18 bytes of the procedure name. We recommend creating a SQL Procedure using a method such as DBSQLPR. We support but do not recommend creating an SQL procedure from a preprocessed program.

See the note about language conformance in [External Procedures](#) (see page 622).

**Note:** For all except LANGUAGE SQL, the program source code for the procedure must be preprocessed before the CREATE PROCEDURE statement is allowed to run.

**Valid Entries:**

COBOL, PLI, C, ASSEMBLER, SQL

**Default Value:**

(No default)

### **SPECIFIC name**

Part of the procedure attributes definition.

ANSI supports *overloading* of the procedure name, with duplicate procedure names resolved to procedure definitions using the layout of the parameter list. For syntax compatibility purposes, a second, *specific* name may be given to a procedure to uniquely identify it, but it must match the nonspecific name, and the nonspecific name must be unique.

### **DETERMINISTIC/NOT DETERMINISTIC**

Part of the procedure attributes definition. This is informational only, indicating whether the procedure always returns the same output, given a certain input.

### **PARAMETER STYLE**

Defines how parameters are passed into and out of the procedure program you write, and how errors are handled.

**Note:** Parameter style applies only to External Procedures. It does not apply to SQL Procedures.

See [Parameter Styles and Error Handling](#) (see page 80) for detailed information.

You can specify PARAMETER STYLE only once, either as:

- Part of the optional *proc-attributes* (procedure attributes) syntax that is specified before the EXTERNAL keyword as shown in the CREATE PROCEDURE *proc-attributes* syntax
- Part of the optional parameter style definition following the EXTERNAL keyword as shown in the CREATE PROCEDURE syntax in [CREATE PROCEDURE Syntax and Description](#) (see page 624).

Parameter style applies only to External Procedures. It does not apply to SQL Procedures.

### **PARAMETER STYLE GENERAL**

This parameter style means 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 SQL's internal SQLCODE variable following the last SQL request made by the procedure. This also applies to parameter style GENERAL WITH NULLS (see following).

### **PARAMETER STYLE GENERAL WITH NULLS**

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

### PARAMETER STYLE 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 (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 (a 4-byte integer) containing the CA Datacom/DB internal error code on output. You cannot have spaces in the field, so it must be 0000 (hex 'FOFOFOFO') if no internal error is returned. Or if, for example, a return code 94 is returned with an internal error code of 100, hex 'FOF1FOFO' would be returned. Your code should convert the binary one-byte field to a four byte unsigned display number.

### PARAMETER STYLE SQL

When a procedure is created using PARAMETER STYLE SQL in the CREATE PROCEDURE statement, the SQLSTATE status indicator is returned in the SQLCA. For detailed information about the SQLSTATE status indicator, see the *CA Datacom/DB Message Reference Guide*.

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 Reerii, 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.

**Valid Entries:**

GENERAL, GENERAL WITH NULLS, DATACOM SQL, SQL

**Default Value:**

DATACOM SQL

How procedure errors are handled depends on the PARAMETER STYLE specified in the CREATE PROCEDURE statement. See the parameter style information in the error handling section in [Parameter Styles and Error Handling](#). (see page 80)

***data-access***

Data access is part of the *proc-attributes* (procedure attributes) definition. If you are using SQL Procedures, *data-access* is ignored.

**Valid Entries:**

MODIFIES SQL DATA, READS SQL DATA, CONTAINS SQL, NO SQL

**Default Value:**

(No default)

***proc-SQL-stmt***

A *proc-SQL-stmt* is part of the *proc-body* syntax. The *proc-SQL-stmt* is composed of SQL Procedure statements that are coded and executed directly. Use the compound statement to create program-like logic. For details about compound statements, see [Compound Statement](#) (see page 642).

In statements that are within SQL Procedures, that is, not External Procedures, SQL parameters and SQL variables can be referenced anywhere expressions are allowed. Colons are not allowed in these references. When a parameter or variable is referenced in a context containing or within the scope of an identically named variable or column, you must disambiguate the name as follows:

For parameters, use the following:

authid.SQL-proc-name.SQL-parameter-name

The *authid* is optional if the procedure name is unique in this particular context. For variables, use the *start-label* of the compound statement as follows:

`start-label.variable-name`

If your procedure fails to avoid ambiguity as required, CA Datacom/DB attempts to resolve the ambiguity in the following way. If the statement involves a table or view, an attempt is made to find a matching column name. All currently executing contexts are next searched for a matching SQL variable, inner-most context first (inner-most active compound-statement, including active condition handlers). The inner-most SQL parameter list is then searched.

We recommend you do not label a compound statement using the SQL Procedure name. If you do so, know that we do not guarantee any specific result or its consistency over time.

Recursive procedure calls are supported. Each occurrence has separate copies of parameters and SQL variables.

The statements in a *proc-SQL-stmt* are as follows:

***executable-DML-stmt***

The *executable-DML-stmt* is any DML (Data Manipulation Language) statement, except for DECLARE CURSOR. Use DML statements to access and manipulate data in SQL tables.

***DDL-stmt***

The *DDL-stmt* (Data Definition Language statements) are used to define SQL objects such as tables and views. DDL statements are only allowed when they do not interfere with the preparation, execution, or rebinding of your procedure.

***control-stmt***

Part of the *proc-SQL-stmt* clause that is used only with SQL Procedures.

Following are the control statements you can choose:

**call-proc** Part of the *control-stmt* in the *proc-SQL-stmt* that is used only with SQL Procedures.

**execute-proc** Part of the *control-stmt* in the *proc-SQL-stmt* that is used only with SQL Procedures.

**proc-only-control-stmt** Part of the *control-stmt* in the *proc-SQL-stmt* that is used only with SQL Procedures. The *proc-only-control-stmt* can be a compound statement (*compound-stmt*) or compound statement control statements (*compound-stmt-only-control-statements*).

**compound-stmt** For information about compound statements, see [Compound Statement](#) (see page 642).

**compound-stmt-only-control-statements** Compound statement only control statements can appear only within the context of a compound statement. They can be nested. They can appear immediately inside a compound statement, or inside other statements that are themselves contained by a compound statement. Choices include the following statements:

assignment-stmt (see [Assignment Statement](#) (see page 638))

case-stmt (not a case-expression, see [CASE Statement](#) (see page 640))

if-then-stmt (see [IF-THEN Statement](#) (see page 652))

iterate-stmt (see [ITERATE Statement](#) (see page 653))

leave-stmt (see [LEAVE Statement](#) (see page 654))

loop-stmt (see [LOOP Statement](#) (see page 655))

while-do-stmt (see [WHILE Statement](#) (see page 675))

repeat-stmt (see [REPEAT-UNTIL Statement](#) (see page 658))

Notice the recursive nature of these definitions. Each of these *proc-only-control-stmt* statements qualify indirectly as a *proc-SQL-stmt* (each *proc-only-control-stmt* is a *control-stmt*), but many of them can also have lists of *proc-SQL-stmt* statements embedded in them. This means that the various statement types may be nested within each other to an unlimited depth. Notice however the following limitations.

DDL statements (Data Definition) are only allowed when they do not interfere with the preparation, execution, or rebinding of your procedure. For example, if your procedure creates a table and then uses it, the statements that use the table might fail with TABLE NOT FOUND or a similar error during the table resolution process of the prepare phase, unless that table is manually created before the CREATE PROCEDURE is prepared (it can be dropped afterwards). If your procedure uses and then ALTERs or DROPs a table, the procedure could be marked invalid as it executes, then attempt an auto-rebind (which could fail, depending on the change you made and how the table is used by the procedure).

#### ***diagnostics-stmt***

Part of the *proc-SQL-stmt* clause that is used only with SQL Procedures. See the GET DIAGNOSTICS statement on [GET DIAGNOSTICS Statement](#) (see page 649).

#### **EXTERNAL**

Begins the external body syntax that can be used to specify a name (NAME *ext-name*) for the program you write that constitutes the body of the procedure (see following for more information on NAME *ext-name*). EXTERNAL is only used for External Procedures, not for SQL Procedures.

A parameter style can also be specified as part of the EXTERNAL syntax but only if you have not specified a parameter style as part of the *proc-attributes* (procedure attribute) syntax. Parameter style is only used for External Procedures, not for SQL Procedures.



**NAME *ext-name***

*(Optional)* If you specify EXTERNAL NAME *ext-name*, the external name you specify as the *ext-name* must match both the *load-module* name and PROGRAM-ID (CA Datacom Datadictionary PROGRAM entity-occurrence name) of the program. We recommend you specify a NAME *ext-name* to avoid confusion. NAME **ext-name** is only used for External Procedures, not for SQL Procedures.

**Valid Entries:**

A valid external name that meets the listed requirements

**Default Value:**

If you do not specify a NAME, the SQL name of the procedure is used as the default which in this case must be unique even without an AUTHID, and must be a valid load-module name.

**RUN OPTIONS *run-options-string***

*(Optional)* Specifying RUN OPTIONS *run-options-string* provides control over the environment in which the procedure executes. The *run-options-string* is a quoted string constant of up to 128 characters in length containing run options for use by the IBM OS/390 Language Environment (LE), which controls procedure execution. Following is an example:

```
RUN OPTIONS 'MSGFILE(SYSOUT,FBA,121,0,ENQ)'
```

Using the MSGFILE option prevents unmanaged contention for the SYSOUT data set (SYSLST in z/VSE) from procedures running concurrently. The ENQ parameter allows competing procedure executions (running in distinct LE environments) to share the data set by single-threading access.

The destination of print statements (for example, printf in C or DISPLAY in COBOL), executed in procedures across the Multi-User Facility, tends to be a single data set, usually DDNAME SYSOUT (SYSLST in z/VSE), identified by the Multi-User Facility startup JCL. For that reason, *we recommend using these statements with caution*. Assuming no print statements have been coded, use of MSGFILE is still useful, however, in enqueueing the messages that LE itself occasionally produces.

**Valid Entries:**

A quoted string constant of up to 128 characters in length containing run options for use by the IBM OS/390 Language Environment

**Note:** Details about the contents of the string are documented in the IBM *OS/390 Language Environment for OS/390 & VM Programming Reference*.

**Default Value:**

(No default)

## Diagnostics and Condition Handling

Understanding the concepts of a Diagnostics Area (see following section) and condition handler (see [Condition Handler](#) (see page 635)) provides you with the ability to use the full capabilities of SQL Procedure condition handling.

### Diagnostics Area

A Diagnostics Area is a data structure that represents a list of errors, warnings, and user-defined conditions that have occurred during the execution of a single SQL statement.

SQL maintains an array of Diagnostics Areas, known as a Diagnostics Area Stack, for each logical unit of work (LUW). At any given time, the stack represents the error status of both the currently-active user request and any related condition handler executions (see following) or of the previous request if a request is not currently active.

Each Diagnostics Area contains a header containing information such as a description of the SQL statement that was executed, and an array of condition information areas (CIAs), each area representing a single error, warning, or user-defined condition that occurred during execution of the statement. The number of CIAs that are maintained in each Diagnostics Area is determined by the SQL\_COND\_INFO\_AREAS Multi-User startup option. The amount of memory used by a Diagnostics Area can be computed as follows, where active LUWs is the number of concurrently active LUWs and CIAs is the number of CIAs as specified by SQL\_COND\_INFO\_AREAS:

$$9k * (\text{active LUWs}) * (\text{CIAs})$$

You can obtain an estimate of the number of concurrent SQL LUWs by querying the LUWS column in the SQL\_STATUS (SQS) Dynamic System Table. Assuming you have 50 SQL applications running concurrently, and SQL\_COND\_INFO\_AREAS is 2, then additional memory usage would then become:

$$9k * 50 * 2 = 900k$$

There are four ways to populate a Diagnostics Area. The first way is to simply experience an error or warning in an SQL statement. The second, third, and fourth ways are to execute the statements SIGNAL (see [SIGNAL Statement](#) (see page 663)), RESIGNAL (see [RESIGNAL Statement](#) (see page 660)), and RAISE ERROR (see [RAISE ERROR Statement](#) (see page 657)). To retrieve diagnostics information in your application, execute the GET DIAGNOSTICS statement (see [GET DIAGNOSTICS Statement](#) (see page 649)).

## Condition Handler

A condition handler is a user-written routine embedded in a user-written procedure that detects and acts to recover from errors, warnings, and any user-defined condition that requires action or remediation. When an end-user merges the functionalities provided by condition handlers and the Diagnostics Area Stack, basic error recovery can be provided, with complete sequences of errors available for retrieval, and business-rule related functions can be performed automatically. The condition declaration and condition handler are found in the syntax for the compound statement (see [Compound Statement](#) (see page 642)).

A procedure that, for example, takes customer orders from a phone operator, checks inventory, and reserves warehouse stock to cover them should also, if every item is in stock, ship the orders immediately. A customer, however, could in addition request that individual items be shipped as they become available. The logic of a condition handler could therefore include that situation as follows (for details about condition handling, see [Condition Handler](#) (see page 635)). To implement the procedure just described, do the following:

1. Declare names for the conditions to be handled:

```
DECLARE ALL_ITEMS_IN_STOCK CONDITION;
DECLARE SHIP_AS_AVAILABLE CONDITION;
```

Lists of SQLSTATEs can also be named as conditions using a FOR clause that is not included in this example, but can be seen in the example starting in Sample Procedure 2. General conditions of SQLEXCEPTION, SQLWARNING, and NOT FOUND are automatically defined (see the description of *cond-declarations*).

2. Define the condition handler (see the two examples that follow).

This example sends a message to initiate action by warehouse personnel. In this example, a compound statement is supplied that includes SQL variable references and some logic. A condition handler, however, could also consist of only one SQL statement, as shown in the example that follows this one.

```
DECLARE CONTINUE HANDLER FOR ALL_ITEMS_IN_STOCK, SHIP_AS_AVAILABLE
ship_it: BEGIN ATOMIC
    DECLARE messageLocal VARCHAR(80);
    IF numberItemsAvailable = numberItemsOrdered THEN
        SET messageLocal = 'ORDER IS COMPLETE. SHIP NOW.';
    ELSE
        SET messageLocal = 'SHIP ALL AVAILABLE ITEMS NOW.';
    END IF;
    INSERT INTO warehouseToDo (orderId, instructions)
        VALUES (orderIdParameter, messageLocal);
END end_ship_it;
```

The following example performs essentially the same function as the example just shown. It presents, however, an alternative to the more complicated logic of the previous example by using an INSERT statement coded into the warehouseToDo table.

```
DECLARE CONTINUE HANDLER FOR ALL_ITEMS_IN_STOCK, SHIP_AS_AVAILABLE
    INSERT INTO warehouseToDo (orderId, instructions)
        VALUES (orderIdParameter, 'SHIP ALL AVAILABLE ITEMS NOW');
```

3. When you detect the defined condition, signal this fact to the condition handler. If, for example, the procedure determines that some items are not in-stock, but the client has requested that items ship as they become available, the following statement could be executed to trigger the handler to execute.

```
SIGNAL SHIP_AS_AVAILABLE;
```

**Note:** Condition handlers triggered by particular SQLSTATEs execute automatically and therefore have no need for explicit SIGNAL statements.

For examples of error handlers embedded in procedures, see the example procedure starting in [Sample Procedure 2](#) (see page 667)

## Condition Handlers Optional

Because condition Handlers are optional, you can have the caller of a procedure handle any error condition by not coding any handlers. When an error is then encountered, your procedure backs out any maintenance it performed and aborts. You need handlers for NOT FOUND and SQLWARNING, however, if you do not want completion codes to cause a procedure to abort, for example if you have an SQLSTATE '02000' or SQLCODE 100 at the end of each cursor and (or) warning conditions such as when an SQLSTATE starts with '01' or there is a positive SQLCODE. For an example showing a procedure without handlers, see [Sample Procedure 1](#) (see page 667).

## SQLSTATE and SQLCODE Special Variables

As an alternative to using the GET DIAGNOSTICS statement in a handler to retrieve the SQLSTATE into an SQL variable, and for compatibility with other implementations, supply an SQLSTATE or SQLCODE local variable and code the following as the first statement in your handler:

```
SET SQL-variable-name = sqlstate-or-sqlcode;
```

SQL then assigns a value to your SQLSTATE or SQLCODE variable after every statement execution. Instructions for using this feature follows:

1. Declare SQLSTATE (or SQLCODE) as an SQL variable in your condition handler or a compound statement that contains the handler.
2. As the first (or only) statement in a handler, assign SQLSTATE (or SQLCODE) to your SQL variable. Again, note that if you supply an SQLSTATE (or SQLCODE) variable, Datacom will reset it's value each time a statement is executed, hence the need for a separate SQL variable copy of any value you need to save.

For example, from a functional point of view code example 1 (following on this page) and code example 2 (see [Code Example 2](#) (see page 638)) are equivalent. In the example that follows, logic that is not shown (for space considerations) is indicated by three vertically arranged periods.

### Code Example 1

```
.
.
.
DECLARE sqlState, sqlStateLocal CHAR(5) DEFAULT '00000';
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION, SQLWARNING, NOT FOUND
    SET sqlStateLocal = sqlState;
.
.
.
INSERT INTO orderTable VALUES (customerId, orderId, orderStatus, orderComments);
IF sqlStateLocal <-> '00000' THEN
    LEAVE thisCompoundStatement;
END IF;
.
.
.
```

In the example just shown, you could have substituted for every occurrence of sqlCode and sqlCodeLocal the variables sqlState and sqlStateLocal. We do not recommend using SQLSTATE and SQLCODE at the same time, because each statement, including assignment statements, reset both the SQLSTATE and SQLCODE, meaning that the following situation could result. Consider the following two statements:

```
SET sqlCodeLocal = sqlCode;
SET sqlStateLocal = sqlState;
```

If the two statements just shown should be the first and second statements in a handler, SQL variable `sqlStateLocal` would receive '00000' as the (reset) value. Because the condition information in the Diagnostics Area is preserved until the handler succeeds, the following would work as a coding alternative:

```
SET sqlCodeLocal = sqlCode;
GET STACKED DIAGNOSTICS sqlStateLocal = RETURNED_SQLSTATE;
```

Following is code example 2, which is equivalent from a functional point of view, to the previously shown code example 1. In the example that follows, logic that is not shown (for space considerations) is indicated by three vertically arranged periods.

### Code Example 2

```
.
.
.
DECLARE sqlState, sqlStateLocal CHAR(5) DEFAULT '00000';
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION, SQLWARNING, NOT FOUND
    GET STACKED DIAGNOSTICS sqlCodeLocal =
RETURNED_SQLSTATE;
.
.
.
INSERT INTO orderTable VALUES (customerId, orderId, orderStatus, orderComments);
IF sqlStateLocal <-> '00000' THEN
    LEAVE thisCompoundStatement;
END IF;
.
.
.
```

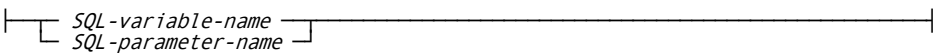
### Assignment Statement

The assignment statement stores a value into an SQL variable or SQL parameter.

Following is the syntax for the assignment statement:

```
►► SET — assignment-target=proc-value-expression ◀◀
```

*Expansion of Where assignment-target is defined as*



**assignment-target**

For the *assignment-target* specify an *SQL-variable-name* or an *SQL-parameter-name*.

**proc-value-expression**

For a description of a *proc-value-expression*, see [CASE Statement](#) (see page 640).

**SQL-variable-name**

The *SQL-variable-name* is the name of a variable that is declared within a compound statement inside a SQL Procedure (a LANGUAGE SQL procedure). If the name conflicts with (matches) another SQL variable name from a nested compound statement (for example, a condition handler), or an SQL parameter name, or a column contained within a referenced table or view, the SQL variable name should be qualified using the start-label of the compound statement that immediately contains it.

**SQL-parameter-name**

The *SQL-parameter-name* is required for SQL Procedures and, though optional for all others, is recommended for all others as well. The *SQL-parameter-name* is the name of a parameter passed to a SQL Procedure (a LANGUAGE SQL procedure). When used in a SQL Procedure containing an SQL variable with a conflicting (matching) name, or a table or view reference where the table or view contains a conflicting column name, the name should be qualified by using the procedure name.

## Example

An assignment statement example follows. In the example that follows, logic that is not shown (for space considerations) is indicated by three vertically arranged periods.

```
.  
. .  
.  
DECLARE mySalary, yourSalary decimal(15,0) DEFAULT 1000000.;  
DECLARE errorClass char(2);  
DECLARE sqlStateLocal char(5);  
. .  
.  
SET errorClass = SUBSTR(sqlStateLocal, 1, 2);  
. .  
.  
SET mySalary = yourSalary * 2;  
. .  
.
```

## CASE Statement

The CASE works in the same way an IF statement works to control which SQL statements, based on predicates you specify, are executed.

Following is the syntax for the CASE statement:

```

▶▶ CASE [ proc-value-expression - simple-when-stmt-list
        | searched-when-stmt-list ]
      [ ELSE - proc-SQL-stmt-list ] END CASE
  
```

*Expansion of Where simple-when-stmt-list is defined as*

```

| WHEN - proc-value-expression - THEN - proc-SQL-stmt-list
  
```

*Expansion of Where searched-when-stmt-list is defined as*

```

| WHEN - proc-search-condition - THEN - proc-SQL-stmt-list
  
```

*Expansion of Where proc-SQL-stmt-list is defined as*

```

| proc-SQL-stmt - ;
  
```

**Note:** For *proc-SQL-stmt* syntax, see the syntax fragment in [CREATE PROCEDURE Syntax and Description](#) (see page 624).

### ***proc-value-expression***

A *proc-value-expression* is a variant form of the expressions used to specify values as follows:

- To the syntax for expressions, the *proc-value-expression* adds SQL variable names (see *SQL-variable-declarations* in [Compound Statement](#) (see page 642)) and SQL parameter names (see *SQL-parameter-name* in [CREATE PROCEDURE Syntax and Description](#) (see page 624)). To view SQL variable names and SQL parameter names in the syntax diagram for expressions, see Expressions.
- A *proc-value-expression* does not allow the use of column functions.  
**Note:** Scalar functions, however, are allowed.
- A *proc-value-expression* does not allow the use of column names.
- A *proc-value-expression* does not allow the use of host variables.

### ***simple-when-stmt-list***

A *simple-when-stmt-list* is a list of simple-when statements, each terminated by a semicolon. For more information, see the example that follows.

### ***searched-when-stmt-list***

A *searched-when-stmt-list* is a list of searched-when statements, each terminated by a semicolon. For more information, see the example that follows.



***proc-search-condition***

The *proc-search-condition* specifies a condition that is true, false, or unknown about a row. The *proc-search-condition* is similar to the *search-condition* described in with the following modifications that allow its use in SQL Procedures:

- SQL parameter and SQL variable references are allowed instead of column and host-variable references.
- Aggregate functions and any predicate involving a sub-query are not allowed.

***proc-SQL-stmt-list***

A *proc-SQL-stmt-list* is a list of *proc-SQL-stmt* statements, each terminated by a semicolon.

## Examples

Following is an example that uses a simple-when statement:

```
CASE creditStatus
  WHEN 'CURRENT' THEN
    call creditLimitCheck(clientId, creditAmount, creditLimit, orderStatus);
    IF (orderStatus = 'CREDIT LIMIT EXCEEDED')
      SET orderStatus = orderStatus CONCAT ', DISCUSS CREDIT LINE INCREASE';
    END IF;
  WHEN 'NEW APPLICATION' THEN
    set orderStatus = 'ACCEPT ORDER PENDING ACCOUNT SETUP COMPLETION';
  WHEN 'PAST DUE' THEN
    set orderStatus = 'NOTIFY CUSTOMER, HOLD FOR PAYMENT RECEIPT';
  WHEN 'BAD' THEN
    set orderStatus = 'REJECTED FOR FAILURE TO PAY';
END CASE;
```

Following is an example that uses a searched-when statement:

```
CASE
  WHEN creditStatus = 'CURRENT' OR clientStatus = 'PRE-APPROVED' THEN
    call creditLimitCheck(clientId, creditAmount, creditLimit, orderStatus);
    IF (orderStatus = 'CREDIT LIMIT EXCEEDED')
      SET orderStatus = orderStatus CONCAT ', DISCUSS CREDIT LINE INCREASE';
    END IF;
  WHEN creditStatus = 'NEW APPLICATION' THEN
    set orderStatus = 'ACCEPT ORDER PENDING ACCOUNT SETUP COMPLETION';
  WHEN creditStatus = 'PAST DUE' THEN
    set orderStatus = 'NOTIFY CUSTOMER, HOLD FOR PAYMENT RECEIPT';
  ELSE
    set orderStatus = 'CREDIT IS ' CONCAT orderStatus;
END CASE;
```



Expansion of Where condition-declarations are defined as

```

| DECLARE - condition-name - CONDITION ----->
|
| FOR - SQLSTATE - VALUE - character-constant | ; ----->

```

Expansion of Where condition-handler is defined as

```

| DECLARE | CONTINUE | HANDLER FOR | , | condition-name | ----->
|         | EXIT      |                                     |   | sqlstate-value |
|         | UNDO      |                                     |   | SQL EXCEPTION  |
|         |                                     |   | SQL WARNING    |
|         |                                     |   | NOT FOUND      |
|
| proc-SQL-stmt - ; ----->

```

Expansion of Where proc-SQL-stmt-list is defined as

```

| proc-SQL-stmt - ; ----->

```

Following are descriptions of the parts of the compound statement syntax:

#### **start-label: / end-label**

(Optional) A *start-label*: is an SQL identifier (followed by a colon) that can be used in various flow-control statements to mark the destination of a branch. When you specify an *end-label* you must also specify a matching *start-label*.

We recommend you do not label a compound statement using the SQL Procedure name. When the SQL Procedure name is used, we cannot guarantee any specific result or its consistency over time.

#### **ATOMIC**

Specifying ATOMIC prevents a procedure from containing a COMMIT or a non-savepoint ROLLBACK. Specify ATOMIC after BEGIN and specify an END before the *end-label*.

#### **SQL-variable-declarations**

(Optional) In *SQL-variable-declarations* each variable receives the default value on entry to a compound statement. Variables without defaults are assigned NULL values on entry. Each declaration can include a list of variables separated by commas. The *variable-name* is an SQL identifier.

The scope (range of visibility) of these variables is the BEGIN-END pair, including any compound statements contained within and any error handlers that execute while the contained compound statement is executing.

Identical variable names can be disambiguated using *start-label.variable-name* where the start label is the *start-label* of a compound statement.

**DEFAULT *literal*/NULL/USER/SYSTEM USER**

(Optional) The DEFAULT clause, an optional part of the *SQL-var-declarations* clause, specifies a default from among the choices that follow:

***literal***

Specifies the default to be a literal. Be certain the literal agrees with the data type of the column. The specified literal can be up to 20 bytes long or the length of the column involved, whichever is shorter. You can specify a default value for a character column where the column is greater than 20 bytes long, but the default literal itself is limited to 20 bytes with the remaining bytes padded with blanks by the system.

**NULL**

Specifies the default as NULL.

**USER**

Specifies the default to be the current authorization ID.

**SYSTEM USER**

Specifies the default to be the accessor ID of the currently signed-on user.

***condition-declarations***

(Optional) Each of the *condition-declarations* relates a name to an SQLSTATE or any user-signaled condition (see the information on the SIGNAL statement on [SIGNAL Statement](#) (see page 663)). A condition handler can be written for any declared condition, including those defined by the user and not related to any SQLSTATE. Declared conditions are visible to all compound statements executing within their context. For instructions and examples of conditions and handlers, see [Diagnostics and Condition Handling](#) (see page 634).

**Note:** The names NOT FOUND, SQLWARNING, and SQLEXCEPTION are pre-defined conditions assigned to SQLSTATE subclasses 02, 01, and any other non-00 subclass, respectively.

**condition-handler**

(Optional) The *condition-handler* defines an SQL statement (a *proc-SQL-stmt* that can be a compound statement) to be executed when a given condition occurs. If a condition occurs for which multiple handlers have been written, SQL chooses and executes the most appropriate handler, after searching available handlers in the inner-most context first, as follows.

1. If an SQLSTATE other than 00000 has occurred and a handler exists in any active context for a condition defined for that specific SQLSTATE, that handler executes. This includes situations where the SQLSTATE was signaled by value or using a *condition-name* associated with a specific SQLSTATE as described in the information about the SIGNAL statement in [SIGNAL Statement](#) (see page 663). A *condition-name* is an SQL identifier.
2. If a condition was signaled using a *condition-name* and the first step did not find a match on SQLSTATE, or if there is no SQLSTATE associated with the *condition-name*, a search is done for a handler associated with the specific *condition-name*.
3. If an SQLSTATE other than 00000 has occurred (including being signaled) and a handler for the generalized condition associated with that SQLSTATE, specifically, SQLEXCEPTION, SQLWARNING, or NOT FOUND, exists, that handler is executed.

If no appropriate handler is found, the SQL statement precipitating the error or warning condition aborts with that condition.

Specific SQLSTATEs and condition names are prevented by CA Datacom from being associated with more than one handler, except when the duplicate handler applies only to a generalized condition under which that SQLSTATE is classified, preventing an ambiguity by following the previously given rules. For instructions and examples of conditions and handlers, see [Diagnostics and Condition Handling](#) (see page 634).

After execution of the handler-action statement(s) without any errors or warnings, the SQLSTATE is reset to 00000 and procedure execution continues as separately described for each of the handler types, CONTINUE, EXIT, and UNDO.

If an error handler generates an unexpected (rather than signaled) error or warning condition, that condition is handled like any other error. To avoid endless loops in error handling, we only handle errors generated in the top level error handler. Unhandled error handler errors cause a RESIGNAL command to execute, followed by an exit from the handler and execution continuation as prescribed by any handler executed on the RESIGNAL. On exit from the handlers, the DIAGNOSTICS AREA STACK (see GET DIAGNOSTICS in [GET DIAGNOSTICS Statement](#) (see page 649)) contains information on any unresolved errors. Unresolved errors cause an abort of the executing procedure statement.

If a SIGNAL, RESIGNAL, or RAISE ERROR statement is executed as part of a handler, CA Datacom/DB assumes that the signaled condition is intended to be seen by the caller of the procedure, and CA Datacom/DB therefore exits with that error intact and unhandled. After successful execution of a CONTINUE type handler, procedure execution resumes with the statement following the one that generated the error.

EXIT handlers operate in a similar way except that when they complete, procedure execution continues after execution of an implied LEAVE statement (see [LEAVE Statement](#) (see page 654)) whose target-label is the one attached to the compound statement in which the handler was defined. The result is that the compound statement containing the statement generating the error aborts but with no error.

UNDO handlers start by performing a ROLLBACK that terminates at the save-point that SQL automatically establishes at the start of the compound statement in which the handler was defined, and closing all cursors open in that context. The handler-action statements then execute. If no unhandled conditions occur, then the condition precipitating the handler execution is cleared, and procedure execution continues after execution of an implied LEAVE statement (see [LEAVE Statement](#) (see page 654)) whose target label is the one attached to the compound statement under which the handler was defined. In other words, the compound statement containing the statement generating the error aborts but with no error.

**Note:** SIGNAL, RAISE ERROR, and RESIGNAL statements that execute inside handlers do not activate additional condition handling. They instead cause the handler to be exited with the signaled condition and cause the execution of the compound statement that triggered the handler to abort. In addition, the compound statement that contains the handler definition is aborted if it is different from the compound statement that triggered the handler, the same as would happen if the handler failed to resolve the triggering condition. The SIGNAL, RAISE ERROR, and RESIGNAL statements should therefore be positioned as the last statements in your condition handler.

#### ***proc-SQL-stmt-list***

A *proc-SQL-stmt-list* is a list of *proc-SQL-stmt* statements, each terminated by a semicolon.

## Example

Following is a compound statement example used in a procedure named `creditLimitCheck`. In the example, the compound statement begins with line number 000017 and ends with line number 000030:

```

000013 -- Does the customer have enough credit to purchase an ordered part?
000014 CREATE PROCEDURE creditLimitCheck
000015   (INOUT result CHAR(80), IN custId CHAR(5), IN purchaseAmount DECIMAL(15,2))
000016   LANGUAGE SQL
000017   limitCheck: BEGIN ATOMIC
000018     DECLARE creditAvailable DECIMAL(15,2) DEFAULT 0;
000019
000020     SELECT creditMax-creditUsed INTO creditAvailable
000021     FROM credit
000022     WHERE credit.custId =
000023           CAST(creditLimitCheck.custId AS NUMERIC(5));
000024
000025     IF (purchaseAmount > creditAvailable) THEN
000026       SET result = 'CREDIT LIMIT EXCEEDED';
000027     ELSE
000028       SET result = 'CREDIT APPROVED';
000029     END IF;
000030   END limitCheck@

```

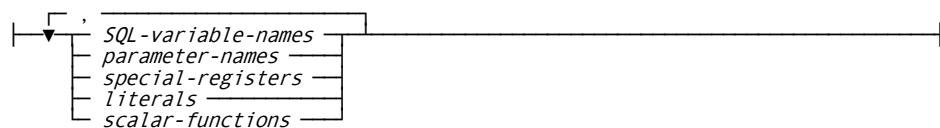
On line 000030 of the previously shown example, the at sign (@), used in conjunction with the `TERM=@` parameter in `DBSQLPR`, enables `DBSQLPR` to skip over semicolons embedded in statements and instead recognize the @ as the end of the statement. For more information about `TERM=`, see [DBSQLPR Options](#) (see page 123).

## DATA COM DUMP Statement

The `DATA COM DUMP` statement can be used as a debugging tool for SQL Procedures. This statement causes requested information to be dumped to a specified file and marked by a header and footer as shown in the first and last lines in the example in [Example](#) (see page 648). The plan name that is shown in the dump report matches the SQL-name of your procedure.

```
▶▶ DATA COM DUMP – dumpable-expression-list – TO PXXSQL ◀◀
```

*Expansion of Where `dumpable-expression-list` is defined as*



### ***dumpable-expression-list***

A list, separated by commas, of *SQL-variable-names*, *parameter-names*, *special-registers*, *literals*, and *scalar-functions*.

**TO PXXSQL**

PXXSQL is the destination supported by the DATACOM DUMP statement. TO PXXSQL causes output to go to the PXX location to which SQL output is sent. The destination of SQL output is either specified using the Multi-User startup option SYSOUT (see the *CA Datacom/DB Database and System Administration Guide*) or allowed to default.

***SQL-variable-names***

The *SQL-variable-names* is the name of a variable that is declared within a compound statement inside a SQL Procedure (a LANGUAGE SQL procedure). If the name conflicts with (matches) another SQL variable name from a nested compound statement (for example, a condition handler), or an SQL parameter name, or a column contained within a referenced table or view, the SQL variable name should be qualified using the start-label of the compound statement that immediately contains it.

***parameter-names***

The *parameter-names* variable refers to any parameter in your procedure.

***special-registers***

For information about [special registers](#) (see page 533).

***literals***

For information about literals, see [literals](#) (see page 510).

***scalar-functions***

For information about scalar functions, see [scalar functions](#). (see page 554)

**Note:** When using scalar functions, operands can include SQL variables and parameters but not column names. Fetching the value of a column into a variable is a suggested way to use a scalar function.

## Example

If you add a tracesOn input parameter or SQL variable to your procedure, you can use it to turn DATACOM DUMP statements on and off, preventing your having to recode the statements if procedure modifications require debugging. For example:

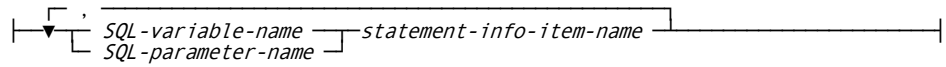
```
IF tracesOn = 'TRUE' THEN
  DATACOM DUMP dumpable-expression-list TO PXXSQL;
END IF;
```

DATACOM DUMP produces output that goes to the PXXSQL and is printed in the detailed format shown in the following sample. In this context, the dumping of constants, for example VALUES ON EXIT in the sample, can help find the output on your test or development MUFs. We reserve the right, however, to change the format of dumps without notice. For example, a less detailed, summary style dump that produces simpler output could be provided at a later time, a summary dump style that could even, at that time, become the default style.

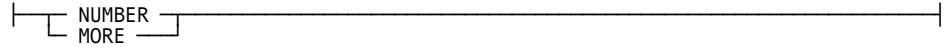




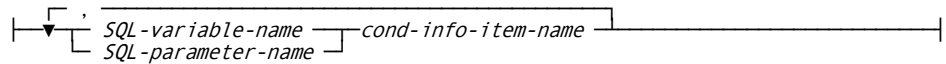
Expansion of Where statement-info-item is defined as



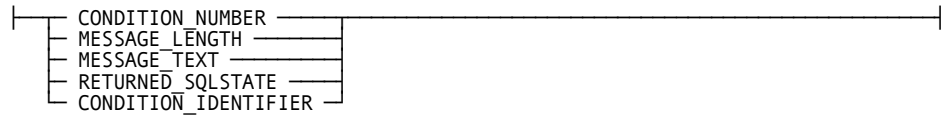
Expansion of Where statement-info-item-name is defined as



Expansion of Where condition-info-item is defined as



Expansion of Where cond-info-item-name is defined as



**CURRENT**

(Optional) Specifying CURRENT (or omitting this optional specification), to indicate which diagnostics area to use, causes information to be retrieved on the SQL error (or other condition) generated by the SQL statement that was just executed.

**STACKED**

(Optional) Specifying STACKED inside an error handler retrieves the Diagnostics Area that is related to the condition that caused the handler to be executed. This original Diagnostics Area has been pushed in order to supply an empty current Diagnostics Area for use by any conditions that can be caused by the error handler itself.

**CONDITION integer-expression**

(Optional) CONDITION followed by an integer that can range in value from one (1) to the number of condition areas available, specifies which condition area to retrieve. Within each Diagnostics Area lies a stack of condition information areas, allowing each Diagnostics Area to store information related to any sequence of errors or conditions that could have occurred during the execution of a single statement.

The number of condition areas available is specified by using the Multi-User startup option SQL\_COND\_INFO\_AREAS (for Multi-User startup option information, see the *CA Datacom/DB Database and System Administration Guide*).

If you do not specify CONDITION, the default is CONDITION 1, which specifies using the most recent error condition, that is, the error condition that occurred during the last non-diagnostics related SQL statement that executed.

***diagnostic-info-items***

The *diagnostic-info-items* variable specifies the information you want to store and where to store it.

***statement-info-item***

The *statement-info-item* contains information about the statement that was executing when the condition occurred.

***statement-info-item-name***

A *statement-info-item-name* (the data type listed in parentheses must be compatible with the data type of the variable or parameter into which the value is being stored) can be one of the following:

NUMBER (SMALLINT) NUMBER indicates the number of condition areas that are in use.

MORE (CHAR(1)) A letter Y (for YES) indicates that more conditions than condition areas occurred.

***condition-info-item***

The *condition-info-item* variable contains information about the condition that occurred.

***cond-info-item-name***

A *cond-info-item-name* (the data type listed in parentheses must be compatible with the data type of the variable or parameter into which the value is being stored) can be one of the following:

CONDITION\_NUMBER (SMALLINT) indicates the condition for which you want to retrieve information. Specifying 1 (a number one) indicates the most recent condition, a 2 specifies the second most recent condition, and so on.

MESSAGE\_LENGTH (SMALLINT) indicates the length of the message text information item.

MESSAGE\_TEXT (VARCHAR(128)) indicates the text of the error message or signal condition.

RETURNED\_SQLSTATE (CHAR(5)) indicates the condition that occurred.

CONDITION\_IDENTIFIER (VARCHAR(128)) gives the name, if one was signaled, of a named condition.

## Example

A GET DIAGNOSTICS statement example follows. In the example that follows, logic that is not shown (for space considerations) is indicated by three vertically arranged periods.

```

.
.
.
DECLARE errMsg varchar(128);
DECLARE sqlStateLocal char(5) default '00000';
DECLARE continue HANDLER FOR sqlexception, sqlwarning, not found
    GET stacked DIAGNOSTICS
        sqlStateLocal = RETURNED_SQLSTATE, errMsg = MESSAGE_TEXT;
.
.
.
startRepeat:
REPEAT
    FETCH orderCrs INTO orderId, custId, creditReqAmt;
.
.
.
UNTIL sqlStateLocal <-> '00000'
END REPEAT DATACOM LOOPLIMIT 100;
.
.
.

```

## IF-THEN Statement

The IF-THEN statement controls logic execution flow in a compound statement, that is, it allows you to specify what logic to use, based on some condition that you specify.

Following is the syntax for the IF-THEN statement:

```

▶▶ IF - proc-search-condition - THEN - proc-SQL-stmt-list ───────────────────▶
└─┬─ ELSEIF - proc-search-condition - THEN - proc-SQL-stmt-list ───────────┘
└─┬─ ELSE - proc-SQL-stmt-list ───────────────────┘ END IF ───────────────────▶

```

***proc-search-condition***

The *proc-search-condition* specifies a condition that is true, false, or unknown about a row. The *proc-search-condition* is similar to the *search-condition* described in with the following modifications that allow its use in SQL Procedures:

- SQL parameter and SQL variable references are allowed instead of column and host-variable references.
- Aggregate functions and any predicate involving a sub-query are not allowed.

***proc-SQL-stmt-list***

A *proc-SQL-stmt-list* is a list of *proc-SQL-stmt* statements, each terminated by a semicolon. For information about a *proc-SQL-stmt*, see [CREATE PROCEDURE Syntax and Description](#) (see page 624).

## Example

An IF-THEN statement example follows. In the example that follows, logic that is not shown (for space considerations) is indicated by three vertically arranged periods.

```

.
.
.
IF sqlStateLocal <-> '00000' THEN
    LEAVE loopExample;
END IF;
.
.
.

```

## ITERATE Statement

The ITERATE statement controls looping logic in a compound statement. Use the ITERATE statement to specify when to skip to the next iteration of a loop.

Following is the syntax for the ITERATE statement:

```

▶▶ ITERATE — statement-label —————▶▶

```

***statement-label***

The *statement-label* specifies the *start-label* of a loop, that is, it relates the ITERATE statement to the position at which execution continues.

## Example

An ITERATE statement example follows. This example skips rows we are not interested in. In the example that follows, logic that is not shown (for space considerations) is indicated by three vertically arranged periods.

```
.  
. .  
startWhile:  
  WHILE sqlStateLocal <-> '00000' DO  
    FETCH orderCrs INTO orderId, custId, creditReqAmt;  
    .  
    .  
    IF creditReqAmt = 0 THEN  
      ITERATE startWhile;  
    END IF;  
    .  
    .  
  END WHILE endWhile DATACOM LOOPLIMIT 10000;  
. .  
.
```

## LEAVE Statement

The LEAVE statement in a compound statement specifies that you want to exit the specified statement, which must contain the LEAVE statement.

Following is the syntax for the LEAVE statement:

```
▶▶ LEAVE — statement-label —————▶▶  
statement-label
```

The *statement-label* specifies the *start-label* of the statement that you want to exit.

## Example

A LEAVE statement example follows. This example exits the loop after a non-zero SQLSTATE. In the example that follows, logic that is not shown (for space considerations) is indicated by three vertically arranged periods.

```

.
.
.
DECLARE continue HANDLER FOR sqlexception, sqlwarning, not found
GET stacked DIAGNOSTICS
    sqlStateLocal = RETURNED_SQLSTATE, errMsg = MESSAGE_TEXT;
.
.
.
loopExample:
LOOP
    FETCH orderCrs INTO orderId, custId, creditReqAmt;
    IF sqlStateLocal <-> '00000' THEN
        LEAVE loopExample;
    END IF;
.
.
.
END LOOP loopExample DATACOM LOOPLIMIT 100;
.
.
.

```

## LOOP Statement

The LOOP statement controls looping logic in a compound statement. It provides an unconditional loop that continues until there is an explicit or condition-based (with a carefully designed EXIT or UNDO condition-handler) loop exit. You must therefore be certain to include a LEAVE statement or SIGNAL statement in your design logic. In addition, we recommend that you use the optional DATACOM LOOPLIMIT clause.

Following is the syntax for the LOOP statement:

```

▶▶ [ start-label: ] LOOP - proc-SQL-stmt-list - END LOOP ▶▶
▶▶ [ end-label; ] [ DATACOM LOOPLIMIT - integer-literal ] ▶▶

```

### **start-label: / end-label**

*(Optional)* A *start-label:* is an SQL identifier (followed by a colon) that can be used in various flow-control statements to mark the destination of a branch. When you specify an *end-label* you must also specify a matching *start-label*. The labels must match.

***proc-SQL-stmt-list***

A *proc-SQL-stmt-list* is a list of *proc-SQL-stmt* statements, each terminated by a semicolon. For information about a *proc-SQL-stmt*, see [CREATE PROCEDURE Syntax and Description](#) (see page 624).

**DATACOM LOOPLIMIT**

We recommend that you use DATACOM LOOPLIMIT. DATACOM LOOPLIMIT is a CA Datacom extension that we provide to allow your proactive avoidance of endless loops that might be caused by faulty logic, unexpected or missing column values, or other unforeseen conditions. The Multi-User startup option SQL\_DATACOM\_LOOPLIMIT can be used to force a default limit on each looping statement. An error is produced if the limit is exceeded.

***integer-literal***

Defines how many loops are too many.

**Example**

A LOOP statement example follows. In the example that follows, logic that is not shown (for space considerations) is indicated by three vertically arranged periods.

```
.  
. .  
. .  
DECLARE continue HANDLER FOR sqlexception, sqlwarning, not found  
  GET stacked DIAGNOSTICS  
    sqlStateLocal = RETURNED_SQLSTATE, errMsg = MESSAGE_TEXT;  
. .  
. .  
loopExample:  
  LOOP  
    FETCH orderCrs INTO orderId, custId, creditReqAmt;  
    IF sqlStateLocal <-> '00000' THEN  
      LEAVE loopExample;  
    END IF;  
. .  
  END LOOP loopExample DATACOM LOOPLIMIT 100;  
. .  
. .
```



## RAISE ERROR Statement

The RAISE ERROR statement is used to signal an error or other condition in the manner of the SIGNAL statement. You must provide a valid SQLSTATE value when using the RAISE ERROR statement.

**Note:** SIGNAL, RAISE ERROR, and RESIGNAL statements that execute inside handlers do not activate additional condition handling. They instead cause the handler to be exited with the signaled condition and cause the execution of the compound statement that triggered the handler to abort. In addition, the compound statement that contains the handler definition is aborted if it is different from the compound statement that triggered the handler, the same as would happen if the handler failed to resolve the triggering condition. The SIGNAL, RAISE ERROR, and RESIGNAL statements should therefore be positioned as the last statements in your condition handler.

►► RAISE ERROR – (*signal-value*, *error-message-text*) ◀◀

*Expansion of Where signal-value is defined as*

| *condition-name* |  
| *sqlstate-value* |

The RAISE ERROR statement functions identically to the following SIGNAL statement syntax:

**Note:** For more information on the SIGNAL statement, see [SIGNAL Statement](#) (see page 663).

►► SIGNAL – *signal-value* – SET MESSAGE\_TEXT = – *error-message-text* ◀◀

*Expansion of Where signal-value is defined as*

| *condition-name* |  
| *sqlstate-value* |

### **signal-value**

*(Optional)* When a *signal-value* is specified, existing Condition Areas within the current Diagnostics Area are pushed down in the Condition Area stack, that is, Condition Area number one becomes Condition Area number two. Condition Area number one is then populated with any supplied or implied RETURNED\_SQLSTATE (supplied using *sqlstate-value*) and (or) CONDITION\_IDENTIFIER (supplied using *condition-name*) in addition to any other information supplied in the SET *signal-information* clause. The *statement-information-items* are modified to indicate that a RESIGNAL statement was the last to modify the Diagnostics Area.

**condition-name**

See the *condition-name* information in the compound statement section that begins in [Compound Statement](#) (see page 642).

**sqlstate-value**

See the *sqlstate-value* information in the compound statement section that begins in [Compound Statement](#) (see page 642).

**error-message-text**

The text of an error message whose occurrence you want to signal an error.

Example

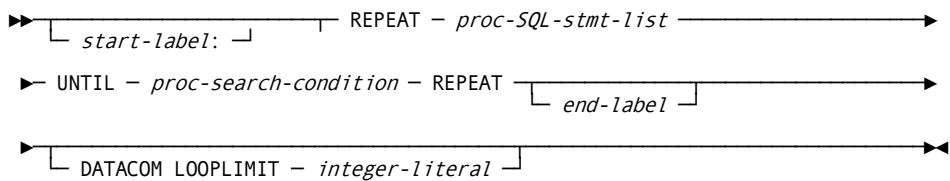
A RAISE ERROR statement example follows. In the example that follows, logic that is not shown (for space considerations) is indicated by three vertically arranged periods.

```
.  
. .  
. .  
IF creditReqAmt > creditAvailable  
  RAISE ERROR ('2FS04', 'CUSTOMER HAS INSUFFICIENT CREDIT LIMIT TO COMPLETE  
PURCHASE');  
END IF;  
. .  
.
```

REPEAT-UNTIL Statement

The REPEAT-UNTIL statement defines conditional looping logic in a compound statement. It provides a means to loop until some predicate becomes true. We recommend that you use the optional DATACOM LOOPLIMIT clause with this statement.

Following is the syntax for the REPEAT-UNTIL statement:



***start-label: / end-label***

(Optional) A *start-label*: is an SQL identifier (followed by a colon) that can be used in various flow-control statements to mark the destination of a branch. When you specify an *end-label* you must also specify a *start-label*, and the labels must match.

***proc-SQL-stmt-list***

A *proc-SQL-stmt-list* is a list of *proc-SQL-stmt* statements, each terminated by a semicolon. For information about a *proc-SQL-stmt*, see [CREATE PROCEDURE Syntax and Description](#) (see page 624).

***proc-search-condition***

The *proc-search-condition* specifies a condition that is true, false, or unknown about a row. The *proc-search-condition* is similar to the *search-condition* described in with the following modifications that allow its use in SQL Procedures:

- SQL parameter and SQL variable references are allowed instead of column and host-variable references.
- Aggregate functions and any predicate involving a sub-query are not allowed.

**DATACOM LOOPLIMIT**

We recommend that you use DATACOM LOOPLIMIT. DATACOM LOOPLIMIT is a CA Datacom extension that we provide to allow your proactive avoidance of endless loops that might be caused by faulty logic, unexpected or missing column values, or other unforeseen conditions. The Multi-User startup option SQL\_DATACOM\_LOOPLIMIT can be used to force a default limit on each looping statement. An error is produced if the limit is exceeded.

***integer-literal***

The *integer-literal* specifies the maximum number of times the loop is allowed to execute.

## Example

A REPEAT-UNTIL statement example follows. In the example that follows, logic that is not shown (for space considerations) is indicated by three vertically arranged periods.

```
.  
. .  
. .  
    DECLARE continue HANDLER FOR sqlexception, sqlwarning, not found  
        GET stacked DIAGNOSTICS  
            sqlStateLocal = RETURNED_SQLSTATE, errMsg = MESSAGE_TEXT;  
. .  
. .  
startRepeat:  
    REPEAT  
        FETCH orderCrs INTO orderId, custId, creditReqAmt;  
. .  
. .  
    UNTIL sqlStateLocal <-> '00000'  
    END REPEAT DATACOM LOOPLIMIT 100;  
. .  
. .
```

## RESIGNAL Statement

The RESIGNAL statement is generally used within a handler to either add information to the condition information items in a Diagnostics Area that represents an error being currently handled, or to fill in subsequent Condition Areas within a Diagnostics Area to indicate that a subsequent error occurred during error handling. The most appropriate handler, if one exists, is also triggered. A RESIGNAL that specifies an SQLSTATE that has no handler can also be used to alter or customize the error information seen by the end-user.

If a handler executes, SQL statement execution might not continue following completion of the SIGNAL statement. The SQL statement that is executed after a SIGNAL completes is dependent upon how the handler was declared and whether that handler succeeds. For example, if a CONTINUE type handler executes and succeeds, execution continues as if no error occurred. For information about other possibilities such as UNDO and EXIT, see [Diagnostics and Condition Handling](#) (see page 634).

**Note:** SIGNAL, RAISE ERROR, and RESIGNAL statements that execute inside handlers do not activate additional condition handling. They instead cause the handler to be exited with the signaled condition and cause the execution of the compound statement that triggered the handler to abort. In addition, the compound statement that contains the handler definition is aborted if it is different from the compound statement that triggered the handler, the same as would happen if the handler failed to resolve the triggering condition. The SIGNAL, RAISE ERROR, and RESIGNAL statements should therefore be positioned as the last statements in your condition handler.

►► RESIGNAL signal-value signal-information ►►

*Expansion of Where signal-value is defined as*

condition-name  
sqlstate-value

*Expansion of Where signal-information is defined as*

SET – signal-info-item-list

*Expansion of Where signal-info-item-list is defined as*

signal-info-item

*Expansion of Where signal-info-item is defined as*

condition-info-item-name – =ssbl. – proc-value-expression

### **signal-value**

*(Optional)* When a *signal-value* is specified, existing Condition Areas within the current Diagnostics Area are pushed down in the Condition Area stack, that is, Condition Area number one becomes Condition Area number two. Condition Area number one is then populated with any supplied or implied RETURNED\_SQLSTATE (supplied using *sqlstate-value*) and (or) CONDITION\_IDENTIFIER (supplied using *condition-name*) in addition to any other information supplied in the SET *signal-information* clause. The *statement-information-items* are modified to indicate that a RESIGNAL statement was the last to modify the Diagnostics Area.

### **signal-information**

*(Optional)* If *signal-information* is supplied without a *signal-value*, that information is copied into Condition Area number one to represent the error condition being handled. The most appropriate handler, if one exists, is then triggered.

**condition-name**

See the *condition-name* information in the compound statement section that begins in [Compound Statement](#) (see page 642).

**sqlstate-value**

See the *sqlstate-value* information in the compound statement section that begins in [Compound Statement](#) (see page 642).

**SET signal-info-item-list**

The SET *signal-info-item-list* specification is used to set a *condition-info-item* such as the error message.

**condition-info-item-name**

For information about the *condition-info-item-name*, including a list of the condition information items that can be set.

**proc-value-expression**

For information about the *proc-value-expression* see [CASE Statement](#) (see page 640).

## Example

A RESIGNAL statement example follows. In this example, SQLSTATE 'S0480' remains unhandled and can be seen by the end user. In the example that follows, logic that is not shown (for space considerations) is indicated by three vertically arranged periods.

```
.  
. .  
declare continue handler for sqlexception  
errHandler1: begin atomic  
    set errCount = errCount + 1;  
    get STACKED diagnostics  
        sqlStateLocal = returned_sqlstate, errMsg = message_text;  
    insert into errTable values  
        (orderId, errCount, "sqlState", errMsg);  
    if sqlStateLocal = 'S0480'  
        RESIGNAL  
    endif  
end errHandler1;  
. . .
```

## SIGNAL Statement

The SIGNAL statement signals an error or other condition, populates the first Diagnostics Area (see [Diagnostics and Condition Handling](#) (see page 634)), and triggers the execution of the appropriate condition handler, if one exists. The information supplied by the SIGNAL statement is available for retrieval by the GET DIAGNOSTICS statement until the error condition is either resolved or a subsequent statement is executed.

The first Diagnostics Area is cleared and the RETURNED\_SQLSTATE and/or CONDITION\_IDENTIFIER are set in the first condition information area of that Diagnostics Area. Certain statement information items such as NUMBER and MORE are also filled in. Any signal information supplied is also moved to the condition information area. The most appropriate error/condition handler (if one exists) is then triggered.

If a handler executes, SQL statement execution might not continue following completion of the SIGNAL statement. The SQL statement that is executed after a SIGNAL completes is dependent upon how the handler was declared and whether that handler succeeds. For example, if a CONTINUE type handler executes and succeeds, execution continues as if no error occurred. For information about other possibilities such as UNDO and EXIT, see [Diagnostics and Condition Handling](#) (see page 634).

**Note:** SIGNAL, RAISE ERROR, and RESIGNAL statements that execute inside handlers do not activate additional condition handling. They instead cause the handler to be exited with the signaled condition and cause the execution of the compound statement that triggered the handler to abort. In addition, the compound statement that contains the handler definition is aborted if it is different from the compound statement that triggered the handler, the same as would happen if the handler failed to resolve the triggering condition. The SIGNAL, RAISE ERROR, and RESIGNAL statements should therefore be positioned as the last statements in your condition handler.

►► SIGNAL – *signal-value* ┌ ──── *signal-information* ──── ────►

*Expansion of Where signal-value is defined as*

┌───┬─── *condition-name* ───┐  
└───┴─── *sqlstate-value* ───┘

*Expansion of Where signal-informatiion is defined as*

┌── SET – *signal-info-item-list* ───┐ ────┘

*Expansion of Where signal-info-item-list is defined as*

┌───┐  
└─── *signal-info-item* ───┘

*Expansion of Where signal-info-item is defined as*

┌── *condition-info-item-name* – = – *proc-value-expression* ───┐ ────┘

***signal-value***

(Optional) When a *signal-value* is specified, existing Condition Areas within the current Diagnostics Area are pushed down in the Condition Area stack, that is, Condition Area number one becomes Condition Area number two. Condition Area number one is then populated with any supplied or implied RETURNED\_SQLSTATE (supplied using *sqlstate-value*) and/or CONDITION\_IDENTIFIER (supplied using *condition-name*) in addition to any other information supplied in the SET *signal-information* clause. The *statement-information-items* are modified to indicate that a SIGNAL statement was the last to modify the Diagnostics Area.

***signal-information***

(Optional) The supplied *signal-information* is copied into Condition Area number one to represent the condition being signaled. The most appropriate handler, if one exists, is then triggered.

***condition-name***

See the *condition-name* information in the compound statement section that begins in [Compound Statement](#) (see page 642).

***sqlstate-value***

See the *sqlstate-value* information in the compound statement section that begins in [Compound Statement](#) (see page 642).

**SET *signal-info-item-list***

The SET *signal-info-item-list* specification is used to set a *condition-info-item* such as the error message.

***condition-info-item-name***

For information about the *condition-info-item-name*, including a list of the condition information items that can be set, see [GET DIAGNOSTICS Statement](#) (see page 649).

***proc-value-expression***

For information about the *proc-value-expression* see [CASE Statement](#) (see page 640).



## Example

A SIGNAL statement example follows. Another SIGNAL statement example can be found in the section labeled SIGNAL orderApproved in Sample Procedure 2. In the example that follows, logic that is not shown (for space considerations) is indicated by three vertically arranged periods.

```

.
.
.
IF creditReqAmt > creditAvailable
  SIGNAL SQLSTATE '2FS04'
  SET MESSAGE_TEXT = 'CUSTOMER HAS INSUFFICIENT CREDIT LIMIT TO COMPLETE PURCHASE';
END IF;
.
.
.

```

## SIMULATE DATACOM PROCEDURE Statement

(DBSQLPR only) The SIMULATE DATACOM PROCEDURE statement provides support in DBSQLPR for SQL variables, user-controlled cursors, and complex program logic.

► SIMULATE DATACOM PROCEDURE — *compound-statement* —►

### **compound-statement**

The compound statement is described in [Compound Statement](#) (see page 642). The start-label of the compound statement is required in the context of the SIMULATE DATACOM PROCEDURE statement and becomes the SQL-name of an actual procedure that is created, executed, and then dropped.

To simulate a procedure that itself has parameters, either simulate the parameters by declaring SQL variables and assigning values or issue a CREATE PROCEDURE followed by a simulated call to that procedure as shown in the following example.

In the following example, SQL-names that do not consist of delimited identifiers are converted to upper-case by SQL. Also, if you need confirmation of the values returned into the ordersAccepted and ordersRejected variables, use the DATACOM DUMP statement (see [DATACOM DUMP Statement](#) (see page 647)) or insert the values into a table, for retrieval outside of the procedure, by subsequent SELECT statements. The ORDERREVIEW procedure called from this example is created by the example SQL procedure that begins in [Examples](#) (see page 667).

```

SIMULATE DATACOM PROCEDURE
supplyVarsForCall: begin atomic
  declare ordersAccepted, ordersRejected int;
  CALL ORDERREVIEW(ordersAccepted, ordersRejected);
end supplyVarsForCall@

```

Following is a SIMULATE PROCEDURE statement that calls the procedure created by the compound statement example in [Example](#) (see page 647). This SIMULATE PROCEDURE statement contains the DATACOM DUMP example shown in [Example](#) (see page 648).

```

000034 simulate datacom procedure
000035 supplyHostVarsForCall: begin atomic
000036   declare errorMessage, result char(80) default 'N/A';
000037   -- The default value for SQL vars is NULL
000038   declare sqlstateLocal char(5);
000039
000040   declare continue handler for not found
000041     set errorMessage = 'CUSTOMER HAS NO CREDIT RECORD';
000042   declare continue handler for sqlexception, sqlwarning
000043     get diagnostics sqlstateLocal = returned_sqlstate,
000044     errorMessage = message_text;
000045
000046   datacom dump 'VALUES ON ENTRY:', result,errorMessage to pxxsql;
000047   call creditLimitCheck(result, '00001', 999.99);
000048   datacom dump 'VALUES ON EXIT:', result,sqlstateLocal,errorMessage to pxxsql;
000049   --
000050   -- In a more comprehensive example, procedure "creditLimitCheck" could have
000051   -- returned an error message and sqlstate through the parameter list, with
000052   -- the inclusion of an SQLEXCEPTION error handler containing a GET DIAGNOSTICS
000053   -- call. This would prevent the caller from having to code their own
000054   -- GET DIAGNOSTICS statement to retrieve the information into SQL variables.
000055 end supplyHostVarsForCall@

```

## Example

A WHILE statement example follows. In the example that follows, logic that is not shown (for space considerations) is indicated by three vertically arranged periods.

```

.
.
.
  DECLARE continue HANDLER FOR sqlexception, sqlwarning, not found
  GET stacked DIAGNOSTICS
  sqlStateLocal = RETURNED_SQLSTATE, errMsg = MESSAGE_TEXT;
.
.
.
startWhile:
  WHILE sqlStateLocal <-> '00000' DO
    FETCH orderCrs INTO orderId, custId, creditReqAmt;
    .
    .
    .
  END WHILE endwhile DATACOM LOOPLIMIT 10000;
.
.
.

```

## Examples

Following are examples that show the use of SQL Procedures.

### Sample Procedure 1

In this example, SQL names that do not consist of delimited identifiers are converted to uppercase by SQL. Also, notice that the at sign (@), used in conjunction with the TERM=@ parameter in DBSQLPR, enables DBSQLPR to skip over semicolons embedded in statements and instead recognize the @ as the end of the statement. For more information about TERM=, see [DBSQLPR Options](#) (see page 123). For z/OS and z/VSE JCL samples, see [Example JCL](#). (see page 133)

**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.
//          REGION=1024K
//JOBLIB      your DD statements go here (see Listing Libraries for CA Datacom Products)
//SQLEXEC    EXEC PGM=DBSQLPR,
//          PARM='authid=sysadm,prtWidth=1500,inputWidth=80,term=@'
//*CAOESTOP DD DUMMY
//SYSUDUMP DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//STDERR DD SYSOUT=*
//STDOUT DD SYSOUT=*
//OPTIONS DD *
/*
//SYSIN DD *
```

```
-- Supply customer credit limits for procedure creditLimitCheck below.
create table credit
  (custId numeric(10),
   creditMax decimal(15,2),
   creditUsed decimal(15,2)) @
insert into credit values (1, 2000., 1000.)@
commit@
--
-- Does the customer have enough credit to purchase an ordered part?
create procedure creditLimitCheck
  (inout result varchar(25), in custId char(5),
   in purchaseAmount decimal(15,2), in turnDebugDumpsOn int)
language sql
limitCheck: begin atomic
  declare creditAvailable decimal(15,2) default 0;

-- Compute available credit.
  select creditMax-creditUsed into creditAvailable
     from credit
     where credit.custId =
           cast(creditLimitCheck.custId as numeric(5));
  if (purchaseAmount > creditAvailable) then
    set result = 'CREDIT LIMIT EXCEEDED';
  else
    set result = 'CREDIT APPROVED';
  end if;

  if (turnDebugDumpsOn = 1) then
    datacom dump purchaseAmount, creditAvailable, result,
              'CREDITLIMITCHECK CUSTID=' || custId
            to pxxsql;
  end if;
end limitCheck@
```

```

--
-- Use SIMULATE to supply a variable for the INOUT "result" parameter
-- SIMULATE was invented to provide procedure-like functionality to DBSQLPR.
simulate datacom procedure
supplyVarsForCall: begin atomic
    declare errorMessage char(80) default 'N/A';
    declare result varchar(25) default 'N/A';
-- The default value for SQL variables is NULL
    declare sqlstateLocal char(5);
--
-- Note that proc. "creditLimitCheck" could have contained the error handlers
-- directly and returned an error message and sqlstate through the parameters.
    declare continue handler for not found
        set errorMessage = 'CUSTOMER HAS NO CREDIT RECORD';
    declare continue handler for sqlexception, sqlwarning
        get diagnostics sqlstateLocal = returned_sqlstate,
            errorMessage = message_text;

    datacom dump 'VALUES ON ENTRY:', result, errorMessage to pxxsql;
-- Note that SQL variable "result" is being used like a host variable here.
    call creditLimitCheck(result, '00001', 999.99, 1);
    datacom dump 'VALUES ON EXIT:', result,sqlstateLocal,errorMessage to pxxsql;
end supplyVarsForCall@
--
-- The above procedure, coded using a SELECT rather than a compound statement:
--commit@
-- Does the customer have enough credit to purchase an ordered part?
create procedure creditLimitCheckV2
    (inout result varchar(80), in custId char(5),
    in purchaseAmount decimal(15,2))
language sql
select case
    when creditMax-creditUsed >= purchaseAmount then
        'CREDIT APPROVED'
    else
        'CREDIT LIMIT EXCEEDED'
    end
into result
from credit
where credit.custId =
    cast(creditLimitCheckV2.custId as numeric(5))@

```

```
--
-- Use SIMULATE to supply a host variable for the INOUT "result" parameter
simulate datacom procedure
supplyVarsForCall: begin atomic
  declare errorMessage, result char(80) default 'N/A';
-- The default value for SQL vars is NULL
  declare sqlstateLocal char(5);

  declare continue handler for not found
    set errorMessage = 'CUSTOMER HAS NO CREDIT RECORD';
  declare continue handler for sqlexception, sqlwarning
    get diagnostics sqlstateLocal = returned_sqlstate,
      errorMessage = message_text;

  datacom dump 'VALUES ON ENTRY:', result, errorMessage to pxxsql;
  call creditLimitCheckV2(result, '00001', 999.99);
  datacom dump 'VALUES ON EXIT:', result,sqlstateLocal,errorMessage to pxxsql;
end supplyVarsForCall@
/*
```

### Sample Procedure 2

In this example, SQL names that do not consist of delimited identifiers are converted to uppercase by SQL. Also, notice that the at sign (@), used in conjunction with the TERM=@ parameter in DBSQLPR, enables DBSQLPR to skip over semicolons embedded in statements and instead recognize the @ as the end of the statement. For more information about TERM=, see [DBSQLPR Options](#) (see page 123). For z/OS and z/VSE JCL samples, see [Example JCL](#). (see page 133)

**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.
//            REGION=1024K
//JOBLIB      your DD statements go here
-- Regarding TERM=@ in the following, DBSQLPR normally uses a semicolon to detect
-- the end of an SQL statement, but SQL statements, as sub-statements within
-- compound-statements, are also required to be terminated with semicolons.
-- This presents a challenge for DBSQLPR, because DBSQLPR must detect the end of a
-- compound-statement (without the problem-prone addition of an extraneous SQL parser
-- outside SQL-proper) despite the appearance of additional semicolons. We circumvent
-- this inherent ambiguity by using the TERM=@ specification to provide a different
-- terminating character for the compound-statement. The specification applies to
-- the entire input file (SYSIN).
//SQLEXEC EXEC PGM=DBSQLPR,PARM='AUTHID=SYSADM,TERM=@,prtwidth=500'
//SYSUDUMP DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//STDOUT DD SYSOUT=*
//OPTIONS DD *
inputwidth=80
/*
//SYSIN DD *
-- Set up tables and data for use by the procedure
drop procedure orderReview CASCADE@
drop table errTable@
drop table newOrder@
drop table warehouseToDo@
commit@
create table errTable
  (orderId int,
   errCountCol int,
   sqlStateCol char(5),
   errMsgCol varchar(128))@
create table newOrder
  (orderId int,
   custId numeric(10),
   creditReqAmt decimal(15,2),
   status char(25))@
create table credit
  (custId numeric(10),
   creditMax decimal(15,2),
   creditUsed decimal(15,2))@
create table warehouseToDo
  (orderId integer,
   instructions char(500),
   whenInstructed timestamp)@
```

```
commit@
delete from newOrder@
delete from credit@
insert into newOrder values(1, 1, 501.01, 'NOT REVIEWED')@
insert into newOrder values(2, 2, 902.02, 'NOT REVIEWED')@
insert into credit values(1,750.01, 100.01)@
insert into credit values(2, 750.02, 100.02)@
commit@
--
CREATE PROCEDURE ORDERREVIEW
  (out ordersAccepted int, out ordersRejected int)
  language sql
-- The rest of this procedure consists of a single compound statement
-- labelled "orderReviewMain".
  orderReviewMain:
  begin atomic
-- Non-delimited variables will be uppercased so ABC matches abc.
    declare statusLocal char(25);
    declare creditReqAmt, creditAvail decimal(15,2);
    declare custId numeric(10);
    declare errCount, orderId, I_want_to_leave_the_loop_NOW,
          I_want_to_re_loop_NOW int;
    declare errMsg varchar(128);
    declare sqlStateLocal char(5);
--
    declare orderCrs cursor for
      select orderId, custId, creditReqAmt
      from newOrder
      where status = 'NOT REVIEWED';
-- Conditions and Condition Handler(s)
--
-- This condition is signaled when an order is approved.
    declare orderApproved condition;
--
-- Handlers may consist of a compound or non-compound statement.
-- Non-compound handler examples:
--
    declare continue handler for orderApproved
      insert into warehouseToDo values (orderReviewMain.orderId,
          'ORDER APPROVED. BEGIN MAKE-READY',
          current timestamp);
```



```

declare undo handler for sqlstate '44444', sqlstate value '55555'
    set I_want_to_leave_the_loop_NOW = 1;

declare exit handler for sqlstate '66666'
    set sqlStateLocal = '66666';

declare continue handler for not found
    set sqlStateLocal = '02000';
--
-- Compound handler example:
-- Note the use of "errCount" even though it's declared outside the
-- scope of the errHandler1 compound statement. If errHandler1 had an
-- errCount variable, we'd have to specify "orderReviewMain.errCount"
-- to reference the variable from the outer compound statement. Also,
-- a locally declared copy of "errCount" would have to have its' value
-- reset on each entry to the handler, hence the "outer" reference.
--
-- Note that there ARE duplicate "errMsg" variables in other scopes
-- but we'll resolve to the one inside the error handler's context.
--
-- SQL exceptions '44444', '55555', and '66666' would trigger the
-- handlers for those specific SQLSTATES. But all other SQLSTATES
-- qualifying as SQLEXCEPTIONS will trigger this handler.
--
declare continue handler for sqlexception, sqlstate '12345',
    sqlwarning
errHandler1: begin atomic
    set errCount = errCount + 1;
    get STACKED diagnostics
        sqlStateLocal = returned_sqlstate,
        errMsg      = message_text;
    insert into errTable values
        (orderId,errCount,sqlStateLocal,errMsg);
end errHandler1;
-- Main Logic Start      --
set errCount = 0;
set ordersAccepted = 0;
set ordersRejected = 0;
set I_want_to_leave_the_loop_NOW = 0;
set I_want_to_re_loop_NOW = 0;
delete from errTable;
set sqlStateLocal = '00000';
-- Loop thru new orders and approve or reject each.
open orderCrs;
start_while:
while (sqlStateLocal = '00000') do

```

```
--
--      Note references to SQL variables
--      fetch orderCrs into orderId, custId, creditReqAmt;
--      Note usage of SQL variable orderWatchMain.custId in query
--      select creditMax-creditUsed into creditAvail
--             from credit
--             where credit.custId = orderReviewMain.custId;

--      if (creditReqAmt > creditAvail) then
--             set statusLocal = 'CREDIT LIMIT EXCEEDED';
--             set ordersRejected = ordersRejected + 1;
--      else
--             set statusLocal = 'CREDIT APPROVED';
--             update credit set creditUsed = creditUsed+creditReqAmt
--                    where custId = orderReviewMain.custId;
--             set ordersAccepted = ordersAccepted + 1;
--             SIGNAL orderApproved;
--      end if;
-- Statement included only to demonstrate usage

--      if (I_want_to_re_loop_NOW = 1)
--             then iterate start_while;
--      end if;
--      update newOrder set status = statusLocal
--             where current of orderCrs;
-- Statement included only to demonstrate usage

--      if (I_want_to_leave_the_loop_NOW = 1)
--             then leave start_while;
--      end if;

--      end while end_while_label datacom looplimit 100;
--      close orderCrs;
-- end
-- orderReviewMain@
COMMIT@
```

```

-- Confirm contents
SELECT * from newOrder@
DELETE from errTable@
drop procedure supplyHostVarsForCallStmt@
commit@
SIMULATE DATACOM PROCEDURE
supplyHostVarsForCallStmt: begin atomic
    declare ordersAccepted, ordersRejected int;
    CALL ORDERREVIEW(ordersAccepted, ordersRejected);
    datacom dump ordersAccepted, ordersRejected to pxxsql;
end supplyHostVarsForCallStmt@
--
-- Confirm results
SELECT * from newOrder@
select * from errTable@
select * from warehouseToDo@
/*

```

## WHILE Statement

The WHILE statement defines conditional looping logic in a compound statement. It provides a means to loop while some predicate remains true. We recommend that you use the optional DATACOM LOOPLIMIT clause with this statement.

Following is the syntax for the WHILE statement:

```

▶▶ [ start-label: ] WHILE - proc-search-condition - DO ───────────────────▶
▶ proc-SQL-stmt-list - END WHILE [ end-label ] ───────────────────▶
▶ [ DATACOM LOOPLIMIT - integer-literal ] ───────────────────▶▶▶

```

### ***start-label: / end-label***

(Optional) A *start-label* is an SQL identifier (followed by a colon) that can be used in various flow-control statements to mark the destination of a branch. When you specify an *end-label* you must also specify a *start-label*, and the labels must match.

### ***proc-search-condition***

The *proc-search-condition* specifies a condition that is true, false, or unknown about a row. The *proc-search-condition* is similar to the *search-condition* described in with the following modifications that allow its use in SQL Procedures:

- SQL parameter and SQL variable references are allowed instead of column and host-variable references.
- Aggregate functions and any predicate involving a sub-query are not allowed.

***proc-SQL-stmt-list***

A *proc-SQL-stmt-list* is a list of *proc-SQL-stmt* statements, each terminated by a semicolon. For information about a *proc-SQL-stmt*, see [CREATE PROCEDURE Syntax and Description](#) (see page 624).

**DATACOM LOOPLIMIT**

We recommend that you use DATACOM LOOPLIMIT. DATACOM LOOPLIMIT is a CA Datacom extension that we provide to allow your proactive avoidance of endless loops that might be caused by faulty logic, unexpected or missing column values, or other unforeseen conditions. The Multi-User startup option SQL\_DATACOM\_LOOPLIMIT can be used to force a default limit on each looping statement. An error is produced if the limit is exceeded.

***integer-literal***

The *integer-literal* specifies the maximum number of times the loop is allowed to execute.

## CREATE RULE

For information about the CREATE RULE statement, see [CREATE TRIGGER/RULE](#) (see page 702).

## CREATE SCHEMA

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
CREATE SCHEMA	YES	YES	

**Note:** YES indicates a valid execution method for this statement. For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

The CREATE SCHEMA statement creates a schema which defines an SQL environment. A schema exists for each authorization ID and contains all table, view, plan, and synonym definitions which are qualified by that authorization ID.

When a CREATE SCHEMA statement successfully executes, the name of the schema is defined in CA Datacom Datadictionary as an AUTHORIZATION entity-occurrence.

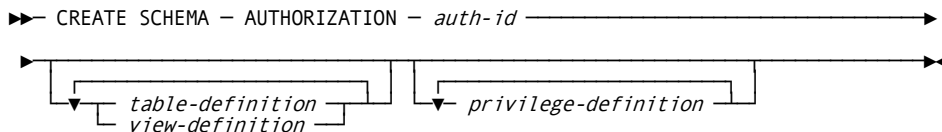
If the CA Datacom/DB Security Facility is installed at your site, you must be a global database owner to execute the CREATE SCHEMA statement.

**Note:** For information about global database owners, see the *CA Datacom Security Reference Guide*.

With regard to table partitioning, CREATE statements may not be issued against a table which is partitioned nor against a partition.

**Note:** For more information about table partitioning, see the *CA Datacom/DB Database and System Administration Guide*.

Following is the syntax diagram for the CREATE SCHEMA statement:



## Description

### **AUTHORIZATION *auth-id***

Specify a unique authorization ID. The authorization ID must be 1 to 18 characters. This rule applies for both ANSI and FIPS modes and the CA Datacom/DB extended mode.

### ***table-definition***

You can define a table using CREATE TABLE. For information about CREATE TABLE see [CREATE TABLE](#) (see page 680).

### ***view-definition***

You can define a view using CREATE VIEW. For information about CREATE VIEW see [CREATE VIEW](#) (see page 705).

### ***privilege-definition***

You can grant privileges using GRANT in databases that have been defined for GRANT in the CA Datacom Datadictionary. For information about GRANT see [GRANT](#) (see page 742).

Example

The following example creates the schema for the authorization ID JOE, and the table named PROJECTTBL.

```
EXEC SQL
    CREATE SCHEMA AUTHORIZATION JOE

    CREATE TABLE PROJECTTBL
        (PROJECTNUM CHAR(6),
        PROJECTNAME CHAR(24),
        DEPTNO CHAR(3),
        EMPNUM CHAR(6),
        PROJSTAFF DECIMAL(5,2),
        PRSTARTDATE DECIMAL(6),
        PRENDDATE DECIMAL(6),
        MAJPROJECT CHAR(6))

END-EXEC
```

CREATE SYNONYM

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
CREATE SYNONYM	YES	YES	YES

**Note:** YES indicates a valid execution method for this statement. To learn about using SQL keywords in CA Dataquery, see the *CA Dataquery User Guide*. For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

The CREATE SYNONYM statement is a CA Datacom/DB extension. CREATE SYNONYM defines an alternative name for a table or view. You can use the synonym to reference a table in another schema without having to enter the qualified name. You can also define synonyms for tables or views in your default schema. When a CREATE SYNONYM statement successfully executes, a SYNONYM entity-occurrence is defined in CA Datacom Datadictionary in PRODUCTION status. The synonym is added to your default schema.

With regard to table partitioning, CREATE statements may not be issued against a table which is partitioned nor against a partition.

**Note:** For more information about table partitioning, see the *CA Datacom/DB Database and System Administration Guide*.

Following is the syntax diagram for the CREATE SYNONYM statement:

```

▶▶ CREATE SYNONYM - synonym - FOR ┌───┐ ┌───┐ ───────────────────▶
                                │   │ │   │
                                │   │ │   │
                                └───┘ └───┘
                                auth-id. table-name
                                view-name

```

## Description

### ***synonym***

The alternative name you want to use when referring to the table or view. The synonym name must not be identical to another synonym or to the unqualified name of a table or view in your default schema. The synonym name can be 1 to 18 characters if you specify the Preprocessor option SQLMODE=ANSI or SQLMODE=FIPS. If SQLMODE=DATACOM, the synonym name can be 1 to 32 characters.

### **FOR**

Introduces the qualified name of the table or view for which you are creating the synonym.

### ***auth-id.***

Specify an authorization ID if you are creating a synonym for a table or view in a schema other than your default schema. Use a period (.) to concatenate the authorization ID to the table or view name (for example, *auth-id.table-name*).

### ***table-name or view-name***

Must name a table or view described in the CA Datacom Datadictionary. The synonym is defined only for your authorization ID, that is to say, the authorization ID of the CREATE SYNONYM statement when the statement is prepared.

## Example

The following example defines an alternative name, OUTDEPTS, for the table DEPTTBL. KWAN is the authorization ID of the user who owns DEPTTBL.

```
EXEC SQL
CREATE SYNONYM OUTDEPTS
FOR KWAN.DEPTTBL
END-EXEC
```

## CREATE TABLE

In the following table, YES indicates a valid execution method for this statement.

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
CREATE TABLE	YES	YES	YES

To learn about using SQL keywords in CA Dataquery, see the *CA Dataquery User Guide*.

For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

**Note:** If you are using CA Datacom/DB as part of the CA Datacom/AD environment, you cannot use the CREATE TABLE statement.

CREATE TABLE defines a table. In this statement, you must specify:

- The name of the table, and
- The column definition for the table.

Optional specifications include:

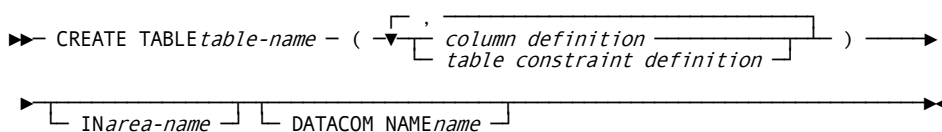
- The table constraint definition for the table.
- Designating more than one column definition.
- Designating one or more table constraint definitions.
- Naming the data area where the table data is to reside.



When a CREATE TABLE statement successfully executes, a TABLE entity-occurrence is defined in CA Datacom Datadictionary in PRODUCTION status, cataloged to the CA Datacom/DB Directory (CXX), and is ready to be populated with data. See Results of Defining Structures Using SQL Statements for information about the results in CA Datacom Datadictionary of using the CREATE TABLE statement.

With regard to table partitioning, CREATE statements may not be issued against a table which is partitioned nor against a partition. An SQL integrity constraint cannot reference a partitioned table, nor a partition of a partitioned table. That is to say, constraints and partitioned tables are mutually exclusive. For more information about table partitioning, see the *CA Datacom/DB Database and System Administration Guide*.

Following is the syntax diagram for the CREATE TABLE statement:



#### Notes:

DATACOM NAME name is a CA Datacom/DB extension.

IN area-name is a CA Datacom/DB extension.

## Description

### **table-name**

The name of the table you are creating. The name you supply, including the implicit or explicit qualifier, must not identify a table, view or synonym already described in the CA Datacom Datadictionary.

If you specify SQLMODE=ANSI or SQLMODE=FIPS in the Preprocessor options, the table name can be 1 to 18 characters in length.

If you specify SQLMODE=DATACOM for extended mode in the Preprocessor options, the table name can be 1 to 32 characters in length.

The qualified form is the name preceded by an authorization ID and a period, for example, auth-id.table-name (but do *not* use SYSADM for the auth-id). If you qualify the table-name, the qualifier designates the schema of the table. If you do not qualify the table-name, the default authorization ID is used as the qualifier. If the CREATE TABLE statement is embedded within a CREATE SCHEMA statement, the authid that qualifies the table must be the same as the authid that follows the AUTHORIZATION keyword.

### **column definition**

See [Column Definition](#) (see page 683) for information on the column definition.

### **table constraint definition**

See [Table Constraint Definition](#) (see page 687) for information on the table constraint definition.

### **IN area-name**

Use this CA Datacom/DB extension to specify the name of the area in which the table data is to reside. The name you specify for the area must be the SQL name, not the CA Datacom Datadictionary occurrence name. The area must already exist in the CA Datacom Datadictionary and be cataloged to the Directory (CXX). If you do not specify the area-name, the table is placed in the default area. If you need to specify an area other than the default, or if the default area is full, see your Database Administrator for names of other areas you can specify when creating a table.

### **DATACOM NAME name**

Use this CA Datacom/DB extension to specify a three-character DATACOM NAME name (a TABLE entity-occurrence in CA Datacom Datadictionary ) for the table. If you do not specify a DATACOM NAME name, CA Datacom Datadictionary generates a name for you. (This is not the CA Datacom Datadictionary entity-occurrence name.)

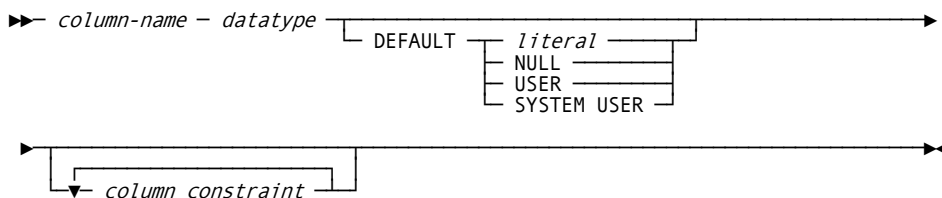
## Privileges

If the Multi-User Facility is secured, whether the creator of the table may access that table depends on the security model of the table. Under the SQL Security Model, when a table is created the user who executes the CREATE TABLE statement is automatically granted all privileges on the table. Under the CA Datacom/DB External Security Model, all access rights, including those for the creator of a table, must be defined through the external security product. See the *CA Datacom Security Reference Guide* for more information.

The CREATE TABLE statement cannot create a remote, partitioned or replicated table. For remote tables, a complete duplicate definition of a remote table in CA Datacom Datadictionary is required for SQL access, and no version control enforcement is available to ensure that remote definitions are synchronized with the local active definition.

## Column Definition

Following is the syntax diagram for the column definition:



### Note:

SYSTEM USER is a CA Datacom/DB extension.

## Description

### **column-name**

The name of a column of the table. Do not use the same name for more than one column in this table.

If you specify `SQLMODE=ANSI` or `SQLMODE=FIPS` in the Preprocessor options, the column name can be 1 to 18 characters in length.

If you specify `SQLMODE=DATACOM` for extended mode in the Preprocessor options, the column name can be 1 to 32 characters in length.

The number of columns you can define for a single table is limited only by the maximum size of the physical record. The sum of the byte counts of the columns must not be less than 1 or greater than 32720.

If you use the `UNIQUE (column-list)` constraint in a table constraint definition (see [Table Constraint Definition](#) (see page 687)), you must separate the last column definition from the `UNIQUE (column-list)` constraint with a comma.

If you specify a `ref-col-name` in the column constraint definition (see [Column Constraint Definition](#) (see page 684)), the data type, length, and scale of the `column-name` specified here in the column definition must be identical to those of the `ref-col-name`.

The first column you name in the `CREATE TABLE` statement becomes the CA Datacom/DB Master and Native Key for the table except when a primary key has been defined. CA Datacom Datadictionary automatically generates this `KEY` entity-occurrence, which has the same name as the column.

**Note:** The `DUPE-MASTER-KEY` and `CHNG-MASTER-KEY` attributes of the `TABLE` entity-occurrence are set to `Y`, indicating that the value of the Master Key can be duplicated and/or changed. The 5-character CA Datacom/DB name of the key is `SQnnn`, where `nnn` is a sequential number unique to each database.

***datatype***

See [Data Types](#) (see page 695).

**Note:** If NOT NULL WITH DEFAULT is specified as a constraint with DATE, TIME, or TIMESTAMP, the default value is the current date, current time, or current timestamp.

**DEFAULT**

Used to specify a default. If DEFAULT is specified, you cannot use a column constraint of WITH DEFAULT.

***literal***

Specifies a literal as the default. The literal you specify must be consistent with the data type of the column. A user-supplied DEFAULT literal can be up to 20 bytes long, or the length of the column involved, whichever is shorter. A default value may be specified for a character column where the column is greater than 20 bytes long, but the default literal itself is limited to 20 bytes, with the remaining bytes padded with blanks by the system.

**NULL**

Specifies NULL as the default. If NULL is specified, you cannot use a column constraint of NOT NULL.

***column constraint***

See the column constraint definition in the following.

**USER**

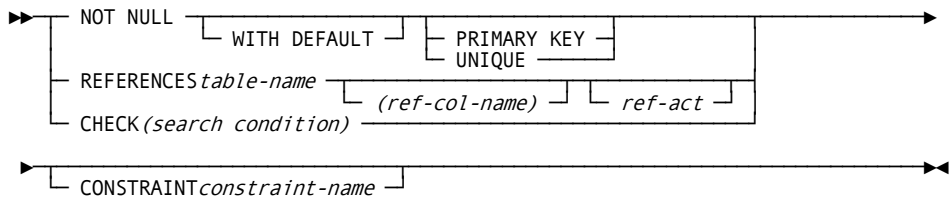
Specifies the current authorization ID as the default.

**SYSTEM USER**

This CA Datacom/DB extension specifies the accessor ID of the currently signed-on user as the default.

Column Constraint Definition

Following is the syntax diagram for the column constraint definition:



**Note:**

WITH DEFAULT is a CA Datacom/DB extension.

---

## Description

### **NOT NULL**

Specifies that the column value cannot be NULL. If NOT NULL is not specified, the column may contain NULL values. On INSERT, a non-null value must be supplied, or a DEFAULT must be specified. On UPDATE, the column cannot be set to the null value. If NOT NULL WITH DEFAULT is specified as a constraint with DATE, TIME, or TIMESTAMP, the default value is the current date, current time, or current timestamp. You must specify NOT NULL to use the column-level UNIQUE constraint. If you specify NOT NULL, you cannot specify NULL as the column default.

### **UNIQUE**

Specifying the column-level UNIQUE constraint after an individual column name indicates that the value for the column is to be unique for each row of the table.

You must specify NOT NULL to use the UNIQUE column-level constraint.

The UNIQUE column-level constraint restricts update or insertion of a row if it contains a column value which has been previously assigned. For example, if COLUMN\_A has been assigned the value '75252' for one row in the table, you cannot insert a row or update another row to assign that same value to the column.

You can specify the column-level UNIQUE on a maximum 50 individual columns per table. The column length must not exceed 180 bytes if you specify this constraint on the column.

When you use the column-level UNIQUE constraint, CA Datacom Datadictionary generates a KEY entity-occurrence with the UNIQUE attribute to enforce uniqueness. This key may not be deleted directly. You must DROP the UNIQUE constraint to delete the key. The CA Datacom Datadictionary name for the key is the same as the column. The 5-character CA Datacom/DB name of the key is SQnnn, where nnn is a sequential number unique to each database.

**PRIMARY KEY**

Same as UNIQUE, except for the following:

- Only one primary key is allowed per table, whether specified at the column or table level.
- When the referential constraint ref-col-list is omitted, the columns of the primary key are implied by default.

**WITH DEFAULT**

This is a CA Datacom/DB extension. Specifying WITH DEFAULT means that if you do not specify a value, blanks is used if the data type is CHARACTER. If the data type is not CHARACTER, zero (0) is used of the appropriate type, length, and scale.

If NOT NULL WITH DEFAULT is specified as a constraint with DATE, TIME, or TIMESTAMP, the default value is the current date, current time, or current timestamp.

You must specify NOT NULL to use the WITH DEFAULT column-level constraint.

Do not use WITH DEFAULT if you specify a DEFAULT in the column definition.

**REFERENCES *table-name***

Specify the name of the table you want to reference. The table-name must identify a table described in the CA Datacom Datadictionary other than a CA Datacom Datadictionary table.

***(ref-col-name)***

If you specify a ref-col-name, the data type, length, and scale of the column-name specified in the column definition (see [Column Definition](#) (see page 683)) must be identical to those of the ref-col-name.

When no ref-col-name is specified, the table-name must have a PRIMARY KEY, and its column-name is the column-name specified in the column definition.

The referenced column must be a UNIQUE or PRIMARY KEY in the referenced table. If the referenced column is not a UNIQUE or PRIMARY KEY, CA Datacom/DB issues a -169 SQL return code.

***ref-act***

Specifies the referential action to be taken when a row in the referenced table is deleted or updated.

**CHECK (*search condition*)**

Allows you to check domain constraints specified in the search condition.

The search condition must be enclosed in parentheses.

When you check a constraint through the CREATE TABLE or ALTER TABLE statements, do not include a function, special register, host variable, subquery, or external table reference in the search condition. If you do, CA Datacom/DB issues an error message informing you that none of those are allowed.

For more information on the search condition, see [Search Conditions](#) (see page 593).

**CONSTRAINT *constraint-name***

Allows you to specify a constraint-name. Constraint names must be unique within all constraints defined in the same schema. A fully qualified constraint name contains the schema ID, that is to say, the authorization ID or creator ID, and a 32-byte name.

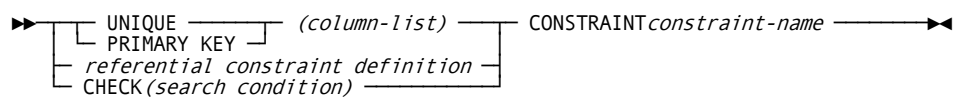
If the name you specify already exists, CA Datacom/DB issues an error message. Query the SYSCONSTRSRC table (see [Schema Information Tables \(SIT\)](#) (see page 821)) to see which names are already in use.

If you do not specify a constraint-name, the system generates a name in the form CONSTRAINT\_nnnn where nnnn is 0001 to 9999. For example, the first constraint-name generated by the system would be CONSTRAINT\_0001, the second would be CONSTRAINT\_0002, and so on.

Constraint names are returned in the SQLCA Error Message to indicate which of possibly several constraints were violated, and in the ALTER TABLE DROP statement to specify which constraint is to be dropped.

## Table Constraint Definition

Following is the syntax diagram for the table constraint definition:



## Description

**UNIQUE**

The UNIQUE table-level constraint specifies that the combination of values assigned to all the listed columns is to be unique.

Individual columns on which the column-level UNIQUE is specified can also be included in the column list for a table-level UNIQUE.

If you use the table-level UNIQUE constraint, CA Datacom Datadictionary generates a KEY entity-occurrence with the UNIQUE attribute to enforce uniqueness. This key may not be deleted directly. You must DROP the UNIQUE constraint to delete the key. The CA Datacom Datadictionary name for the key is the same as the table, followed by a number which makes the name unique. The 5-character CA Datacom/DB name of the key is SQnnn, where nnn is a sequential number unique to each database.

**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.

**PRIMARY KEY**

Same as UNIQUE, except for the following:

- Only one primary key is allowed per table, whether specified at the column or table level.
- When the referential constraint ref-col-list is omitted, the columns of the primary key are implied by default.

**(column-list)**

Specify the column-list for the table-level UNIQUE or PRIMARY KEY constraint by enclosing one or more column names in parentheses. Use a comma to separate column names.

For example, if you specify UNIQUE (COLUMN\_A, COLUMN\_B) when you create a table, the following value assignments are valid because the combination of values for these two columns are unique in each row.

	COLUMN_A	COLUMN_B
Row 1	DALLAS	TEXAS
Row 2	PARIS	TEXAS
Row 3	PARIS	FRANCE

The total length of all columns listed in each table-level constraint cannot exceed 180 bytes.



**referential constraint definition**

See [Referential Constraint Definition](#) (see page 689).

**CHECK (search condition)**

Allows you to check domain constraints specified in the search conditions.

The search condition must be enclosed in parentheses.

When you define a constraint through the CREATE TABLE or ALTER TABLE statements, do not include a function, special register, host variable, subquery, or external table reference in the search condition. If you do, CA Datacom/DB issues an error message informing you that none of those are allowed.

You may define multiple CHECK constraints to indicate exactly which constraint has been violated, since the constraint name is returned in the SQL Error Message.

For information on the search condition, see [Search Conditions](#) (see page 593).

**CONSTRAINT constraint-name**

Allows you to specify a constraint-name. Constraint names must be unique within all constraints defined in the same schema. A fully qualified constraint name contains the schema ID, that is to say, the authorization ID or creator ID, and a 32-byte name.

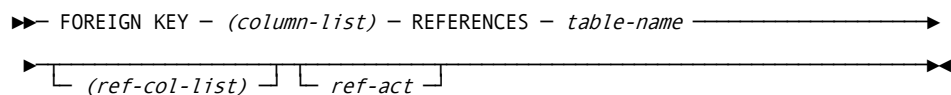
If the name you specify already exists, CA Datacom/DB issues an error message. Query the SYSCONSTSRC table (see [Schema Information Tables \(SIT\)](#) (see page 821)) to see which names are already in use.

If you do not specify a constraint-name, the system generates a name in the form CONSTRAINT\_nnnn where nnnn is 0001 to 9999. For example, the first constraint-name generated by the system would be CONSTRAINT\_0001, the second would be CONSTRAINT\_0002, and so on.

Constraint names are returned in the SQLCA Error Message to indicate which of possibly several constraints were violated, and in the ALTER TABLE DROP statement to specify which constraint is to be dropped.

## Referential Constraint Definition

Following is the syntax diagram for the referential constraint definition:



## Description

**FOREIGN KEY**

Allows you to specify a foreign key.

Foreign keys define relationships between tables. The column(s) of a foreign key in one table are related to the primary or unique key of some table. That related primary or unique key may not be defined on the same table as the foreign key.

Every foreign key is related to a primary or unique key. A primary or unique key may be related to zero, one, or many foreign keys.

Each foreign key defines a referential integrity constraint. Every foreign key value (the column values of the foreign key's columns from a single row of the table) must exactly match a primary or unique key value of the foreign key's related primary or unique key, except when at least one foreign key column value is NULL. When a foreign key value exists which does not match any primary or unique key value of its related primary or unique key, the referential integrity constraint is violated.

The foreign key constraint does *not* restrict the number of rows with the same foreign key value, that is to say, the constraint is "1-to-many," one referenced table row may match multiple referencing rows. You may define a "1-to-1" constraint by also defining the foreign key columns in a primary key or unique constraint.

An attempt to INSERT or UPDATE a foreign key value that does not exist in the referenced primary/unique key is rejected as a foreign key *value* error. An attempt to DELETE or UPDATE a primary/unique value that is referenced by one or more foreign key values is rejected as a foreign key *reference* error.

When the referenced table is loaded or recovered, the integrity of the foreign key is in doubt, and the table is placed in a CHECK-RELATED check state. A table in a check state cannot be opened. You must execute the CONFIRM function of DBUTLTY to confirm that all referenced values still exist. The check state is reported in the Directory (CXX) report.

**Note:** Tables defined using the SQL CREATE TABLE statement have the RECOVERY attribute-type of the TABLE entity-occurrence set to Y (for yes) in CA Datacom Datadictionary.

A "physical" key is *not* generated for a foreign key. However, if you define a key, it is used when checking the DELETE or UPDATE of the referenced primary/unique key. For best performance, the key should include all columns of the foreign key as the only or leading columns. Other columns may be included following the foreign key columns. The order in which foreign key columns are specified in the key does not affect performance.

A foreign key is rejected if the table involved has more than one foreign key and the delete or update actions of any two of those foreign keys conflict. A -36 SQL return code is issued if a conflict occurs. For more information see [Referential Actions That Conflict](#) (see page 693).

**(column-list)**

Use the column-list to specify one or more column names. The column names must be separated by commas and the list must be enclosed in parentheses.

The number of columns in the column-list and their data type, length, and scale must be identical to those specified in the ref-col-list or the default ref-col-list.

When at least one of the columns is not defined with NOT NULL, that is to say, it may contain the null value, the reference is said to be "optional." When at least one column is the null value, no check is made for a matching primary/unique value.

**REFERENCES table-name**

Specify the name of the table you want to reference. The table-name must identify a table described in the CA Datacom Datadictionary other than a CA Datacom Datadictionary table. When no ref-col-list is specified, the table-name must have a PRIMARY KEY, and its column(s) is the default ref-col-list. The referenced table cannot be a remote table. Reference to a table in a different schema is permitted.

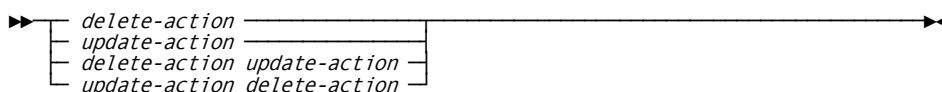
**(ref-col-list)**

Use the column-list to specify one or more column names. The column names must be separated by commas and the list must be enclosed in parentheses. When you are specifying a ref-col-list, specify column(s) of a UNIQUE or PRIMARY KEY constraint on the table-name.

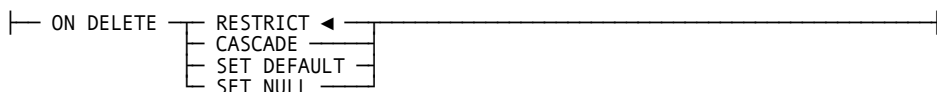
**ref-act**

Specifies the referential action to be taken when a row in the referenced table is deleted or updated.

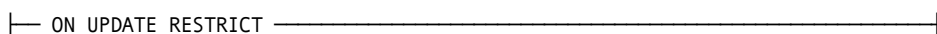
Following is the syntax diagram for ref-act (referential action):



*Expansion of Where delete-action is defined as*



*Expansion of Where update-action is defined as*



### **ON DELETE**

Specifies which delete-action is to be taken. Errors that occur during delete-actions cause the original DELETE (and all propagated delete-actions) to be aborted.

Delete-actions may be one of the following:

#### **RESTRICT**

ON DELETE RESTRICT is the default delete-action. It specifies that there shall be no matching rows when a row is deleted from the referenced table. Matching rows are (for a given row in the referenced table) all rows in the referencing table whose referencing columns equal the corresponding referenced column. If referencing rows exist, the DELETE is rejected with a -175 return code.

#### **CASCADE**

Specifies that referencing rows are deleted after any references to the child table are followed and those referential actions performed.

#### **SET DEFAULT**

Specifies that each column in the referencing foreign key is set to its default value for every referencing row. If ON DELETE SET DEFAULT is specified, all of the columns in the foreign key must have defaults defined.

#### **SET NULL**

Specifies that nullable columns in the referencing foreign key are set to NULL for all referencing rows. At least one column is guaranteed to be nullable in this foreign key. If ON DELETE SET NULL is specified, at least one of the foreign key columns must be nullable.

### **ON UPDATE**

Specifies which update action is to be taken. RESTRICT is the only update action supported. RESTRICT is the default and is enforced even if not specified.

ON UPDATE RESTRICT specifies that there shall be no matching rows when a row is updated in the referenced table. Matching rows are (for a given row in the referenced table) all rows in the referencing table whose referencing columns equal the corresponding referenced column. If referencing rows exist, the UPDATE is rejected with a -175 return code.

## Referential Actions That Conflict

A foreign key is rejected if the table involved has more than one foreign key and the referential actions of any two of those foreign keys conflict. One of two conditions are generally responsible for conflicts between referential actions:

1. A delete on a referenced table can propagate conflicting referential actions to a row of a referencing table. For example, if two foreign keys on table A reference table B, and one foreign key specifies ON DELETE CASCADE but the other specifies ON DELETE SET DEFAULT, then it becomes impossible to update table A in a way that satisfies both foreign keys.
2. The order in which foreign key references are followed and satisfied affects the resulting actions taken on referencing rows. It might initially seem that all situations described in the first condition (previously described) would also be included here. For example, if the SET DEFAULT described in the first condition (previously described) were executed before the CASCADE, the stored default values could effectively sever the row's other foreign key connection to the deleted row in table B, in which case the ON DELETE CASCADE would never be executed. But this second condition can occur even if the delete actions agree.

Consider a case in which both foreign keys on table A specify ON DELETE SET NULL and the columns used in the two keys partially overlap. Setting any of the overlapping columns to NULL would sever both foreign keys' connections, because a NULL in any column of a foreign key severs the connection to the parent row. The first foreign key that is executed would therefore have its columns set to NULL, but the second foreign key would never be executed.

A table listing referential actions that conflict is provided in the following. When referring to the table, please note that it should be interpreted with regard to the following:

- An update action only conflicts with a delete action if the first condition on the previous page is violated, that is to say only if a DELETE on some indirect parent of a table can propagate both an ON UPDATE RESTRICT and a conflicting delete action to the table using two separate foreign key paths at the same time. A DELETE can propagate ON UPDATE RESTRICT because when an ON DELETE SET DEFAULT or ON DELETE SET NULL is executed on a table that is in turn referenced by another foreign key, the update actions on a referencing child table are executed instead of the delete actions.
- Referential actions that would normally conflict are allowed if the foreign keys involved have disjoint sets of parent tables (including both direct and indirect parents), or if the sets of columns affected by the conflicting rules are disjoint. SET NULL and SET DEFAULT actions that are not listed as being in conflict in the following table are, however, rejected if the second condition described on the previous page exists. But that second condition does not apply if the two foreign keys involved contain either totally disjoint or completely matching sets of columns on the child table.

In the following table, the abbreviations describe **CONFLICTS** and represent the following:

**ODR**

Represents **ON DELETE RESTRICT**.

**ODC**

Represents **ON DELETE CASCADE**.

**ODSD**

Represents **ON DELETE SET DEFAULT**.

**ODSN**

Represents **ON DELETE SET NULL**.

**OUR**

Represents **ON UPDATE RESTRICT**.

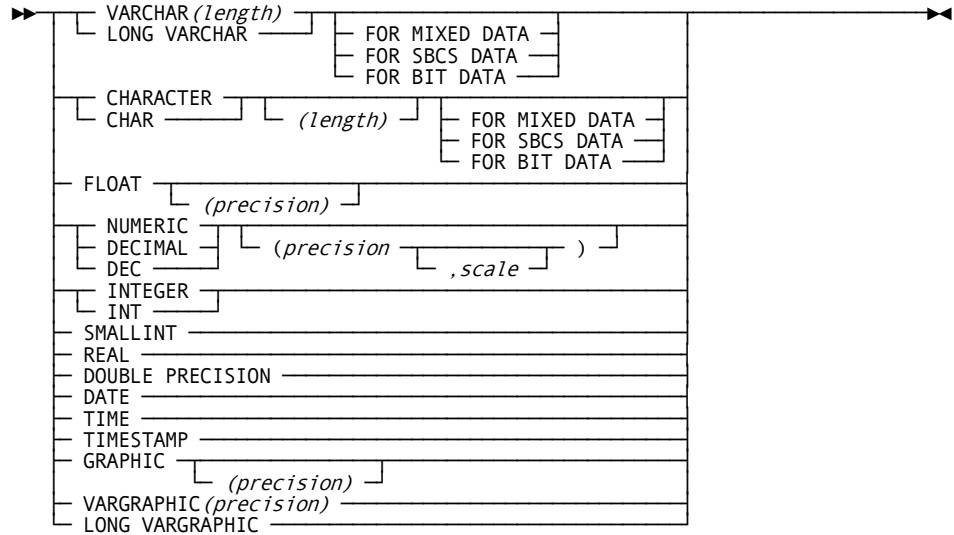
**Note:** The asterisk (\*) means it does not conflict if defaults for all columns involved are NULL.

<b>REFERENTIAL ACTIONS:</b>	<b>ODR</b>	<b>ODC</b>	<b>ODSD</b>	<b>ODSN</b>	<b>OUR</b>
ON DELETE RESTRICT		X	X	X	
ON DELETE CASCADE	X		X	X	X
ON DELETE SET DEFAULT	X	X		X*	X
ON DELETE SET NULL	X	X	X*		X
ON UPDATE RESTRICT		X	X	X	

**Note:** Foreign keys can also be rejected for reasons not mentioned here. For a complete list of reasons, see the SQL return codes section in the *CA Datacom/DB Message Reference Guide*.

## Data Types

Following is the syntax diagram for the valid SQL data types:



The following are CA Datacom/DB extensions:

- VARCHAR
- LONG VARCHAR
- FOR MIXED DATA
- FOR SBCS DATA
- FOR BIT DATA
- DATE
- TIME
- TIMESTAMP
- GRAPHIC
- VARGRAPHIC
- LONG VARGRAPHIC

The following summarizes information about the valid SQL data types. See [Data Types](#) (see page 485) for more information.

**Valid SQL Data Types:**

**CHARACTER (*length*) CHAR (*length*)**

CHARACTER or CHAR specifies a character string. The optional *length* must be an integer designating the number of characters in the string and must be enclosed in parentheses. The default value for the CHARACTER data type length is one byte.

**VARCHAR (*length*)**

VARCHAR specifies a varying-length character string of length 1 to maximum row size. The *length* is required, must be an integer designating the number of characters in the string, and must be enclosed in parentheses.

**LONG VARCHAR**

LONG VARCHAR specifies a varying-length character string whose maximum length is determined by the amount of space available in a block. Use of the LONG VARCHAR Data Type creates rows that are as long as the block size of the containing area. If LONG VARCHARs are to be used, this block size should be taken as the row size to be used in calculating the Log Area (LXX) block size (the block size of the longest block sized area containing LONG VARCHARs should be used).

Three semantic types are allowed: **FOR MIXED DATA**, **FOR SBCS DATA**, and **FOR BIT DATA**. FOR MIXED DATA means that DBCS characters are allowed in values stored in the column, in addition to EBCDIC (Single-Byte Character Set (SBCS)) characters. This is relevant when SQL is processing a value in the column, since it must recognize the Shift-Out and Shift-In characters that delimit DBCS substrings. FOR SBCS DATA means that DBCS characters are not allowed in the column. FOR BIT DATA means that the data is a string of binary data rather than a string of characters.

If a semantic type is not specified when the column is created, the default is the semantic type that was specified on the CXXMAINT OPTION=ALTER,DBC=xx option. If xx was IS or FS, the default is FOR SBCS DATA. If xx was IM or FM, the default is FOR MIXED DATA.

**FLOAT (*precision*)**

Specifies a 64-bit floating-point number.

The optional *precision* of the number is the total number of digits. The *precision* can range from 1 to 15. The *precision* must be enclosed in parentheses.

**NUMERIC (*precision, scale*)**

Specifies a zoned decimal number.

The optional *precision* of the number is the total number of digits. The *precision* can range from 1 to 31. If you specify *precision*, you have the option of also specifying *scale*. The optional *scale* of the number is the number of digits to the right of the decimal point. The *scale* can range from 0 to the precision.



If you specify both *precision* and *scale*, you must separate them with a comma and enclose them in parentheses. When specifying a *scale* of zero, however, you can use (*precision*) for (*precision*,0). If you do not specify *precision* and/or *scale*, the default value used is a *precision* of 5 and a *scale* of zero (5,0).

**DECIMAL (*precision*,*scale*) DEC (*precision*,*scale*)**

Specifies a packed decimal number.

The optional *precision* of the number is the total number of digits. The *precision* can range from 1 to 31. If you specify *precision*, you have the option of also specifying *scale*. The optional *scale* of the number is the number of digits to the right of the decimal point. The *scale* can range from 0 to the *precision*.

If you specify both *precision* and *scale*, you must separate them with a comma and enclose them in parentheses. When specifying a *scale* of zero, however, you can use (*precision*) for (*precision*,0). If you do not specify *precision* and/or *scale*, the default value used is a *precision* of 5 and a *scale* of zero (5,0).

**INTEGER INT**

Specifies a large binary integer with a *precision* of 31 bits.

**SMALLINT**

Specifies a small binary integer with a *precision* of 15 bits.

**REAL**

Specifies a 64-bit floating-point number.

**DOUBLE PRECISION**

Specifies a 64-bit floating-point number.

**DATE**

Specifies a date.

**TIME**

Specifies a time.

**TIMESTAMP**

Specifies a timestamp.

**GRAPHIC (*precision*) / VARGRAPHIC (*precision*) / LONG VARGRAPHIC**

GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC specify Double-Byte Character Set (DBCS) characters. Each character is two bytes long. The *precision* of a DBCS column (optional for GRAPHIC, required for VARGRAPHIC) is the maximum number of two-byte characters that can be stored, not the physical length.

DBCS characters in SQL statements must be delimited by the Shift-Out and Shift-In characters. Shift characters are either IBM-defined (X'0E' and X'0F') or Fujitsu-defined (X'28' and X'29'). Shift characters are specified in CXXMAINT OPTION=ALTER,DBCS=xx in the CA Datacom/DB Utility (DBUTLTY). If xx is IS or IM, the Shift characters are the IBM-defined characters. If xx is FS or FM, the Shift characters are the Fujitsu-defined characters. GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC.

### Example 1

Some examples of using the CREATE TABLE statement are:

Create a table with the following specifications:

1. Authorization ID: JOE

The example assumes that JOE is the default authorization ID in effect when the statement is executed. In this case, it is not necessary to specify the authorization ID.

2. Table name: DEPTTBL
3. Column names: DEPTNO, DEPTNAME, MGRNBR, ADMDEPT
4. Data type: CHAR for all columns
5. Area name: CASQLDEFAULT

The default area name is specified, although it is not necessary. If you do not specify an area name, the table is automatically placed in the default area. You must specify the area name if you want to place the table in an area other than the default.

```
EXEC SQL
    CREATE TABLE DEPTTBL
        (DEPTNO CHAR(2),
         DEPTNAME CHAR(24),
         MGRNBR CHAR(6),
         ADMDEPT CHAR(2))
    IN CASQLDEFAULT
END-EXEC
```

### Example 2

Create a table with the following specifications:

1. Authorization ID: TED

In this example, TED is not the default authorization ID. It is therefore necessary to specify the authorization ID.

2. Table name: DEPTTBL
3. Column names: DEPTNO (value to be unique), DEPTNAME, MGRNBR, ADMDEPT

4. Data type: CHAR for all columns  
EXEC SQL  
CREATE TABLE TED.DEPTTBL  
(DEPTNO CHAR(2) NOT NULL UNIQUE,  
DEPTNAME CHAR(24),  
MGRNBR CHAR(6),  
ADMDEPT CHAR(2))  
END-EXEC

### Example 3

Create a table with the following specifications:

1. Authorization ID: JOE
2. Table name: PROJECTTBL
3. Column names (CHARACTER data): PROJECTNUM (value to be unique), PROJECTNAME (value to be unique), DEPTNO, EMPNUM, MAJPROJECT
4. Column names (DECIMAL data): PROJSTAFF, PRSTARTDATE, PRENDDATE

```
EXEC SQL
CREATE TABLE JOE.PROJECTTBL
(PROJECTNUM CHAR(6) NOT NULL UNIQUE,
PROJECTNAME CHAR(24) NOT NULL UNIQUE,
DEPTNO CHAR(3),
EMPNUM CHAR(6),
PROJSTAFF DECIMAL(5,2),
PRSTARTDATE DECIMAL(6),
PRENDDATE DECIMAL(6),
MAJPROJECT CHAR(6))
END-EXEC
```

### Example 4

Create a table with the same specifications as Example 2, but specify that the DEPTNO and MGRNBR columns are to have a combined value which is unique for this table.

**Note:** The last column definition is separated from the table-level UNIQUE constraint by a comma.

```
EXEC SQL
CREATE TABLE TED.DEPTTBL
(DEPTNO CHAR(2) NOT NULL UNIQUE,
DEPTNAME CHAR(24),
MGRNBR CHAR(6),
ADMDEPT CHAR(2),
UNIQUE (DEPTNO, MGRNBR))
END-EXEC
```

## Example 5

Create a table to keep track of customers who pay their bills on time. Then create another table to keep track of current orders. Design the tables to enforce the following business rules:

1. Never accept an order from a customer who does not pay his bills.
2. Do not finance orders of less than \$1,000.00 worth of merchandise.
3. Always get a down payment of at least 25 percent on financed orders.
4. Never negotiate interest rates to below 11 percent.
5. Ensure maximum performance when retrieving customer orders based on the assigned "order ID."

**Note:** The following example is intended to show the specification of as many different data types and options as possible. For this reason, proper data base design and table normalization have been ignored. Proper design and normalization of your databases is imperative for maximum system performance.

EXEC SQL

```
CREATE TABLE ACCOUNTING.PAYING_CUSTOMERS
(TOTAL_NUM_ORDERS    INTEGER NOT NULL,
 FIRST_ORDER        DATE,
 LAST_ORDER         DATE,
 COMPANY_NAME       CHAR(40) NOT NULL,
 TOTAL_GROSS_ORDERS DECIMAL(12,2) NOT NULL,
 AVG_GROSS_PER_ORDER DECIMAL(9,2) NOT NULL,
 PRIMARY KEY (COMPANY_NAME));
```

END-EXEC

**(Example 5 continued on next page)**

```

EXEC SQL
CREATE TABLE ORDERS
  (ORDER_ID_NUMBER    INTEGER NOT NULL PRIMARY KEY,
   CUSTOMER_NAME      CHAR(40) DEFAULT 'INTERNAL ORDER',
   SHIPMENT_ID_NUMBER DECIMAL(6,0) NOT NULL UNIQUE,
   GROSS_AMOUNT       NUMERIC(8,2) NOT NULL,
   PERCENT_DOWN_PMT   REAL DEFAULT 100.0
                        CHECK (PERCENT_DOWN_PMT >= 25.0),
   NUM_PAYMENTS       SMALLINT DEFAULT NULL,
   INTEREST_RATE      DOUBLE PRECISION CHECK (INTEREST_RATE >= 11.0)
                        CONSTRAINT ORDERS_MIN_INTEREST_RATE,
   DATE_PROMISED      DATE,
   TIME_PROMISED      TIME,
   DATE_AND_TIME_SHIPPED TIMESTAMP NOT NULL WITH DEFAULT,
   AVG_ITEM_QUANTITY  FLOAT,
   AVG_NUM_ITEMS      FLOAT(6),
   TOTAL_PIECES       INTEGER NOT NULL WITH DEFAULT,
   FOREIGN KEY (CUSTOMER_NAME) REFERENCES
                        ACCOUNTING.PAYING_CUSTOMERS(COMPANY_NAME),
   CHECK (GROSS_AMOUNT > 1000.0 OR PERCENT_DOWN_PMT = 100.0)
                        CONSTRAINT ORDERS_MIN_AMT_FINANCED);
END-EXEC

```

#### How the Tables Enforce the Business Rules:

The following numbers correspond to the numbers of the business rules specified at the beginning of this example.

1. The foreign key on ORDERS.CUSTOMER-NAME prevents orders from being inserted into the table ORDERS unless the customer's name appears in the PAYING-CUSTOMERS table in column COMPANY-NAME. Any attempt to insert an order for a nonpaying customer is rejected.
2. Constraint ORDERS-MIN-AMT-FINANCED prevents an order from being inserted if an attempt is being made to finance an order of less than \$1000.00.
3. CHECK (PERCENT-DOWN-PMT >= 25.0) stops orders that are being financed with a down payment of less than 25 percent. Since the user did not specify a constraint name, the system generates a unique name. It is recommended that you give meaningful names to all constraints so that error messages are more meaningful when a constraint name is in the message.
4. Constraint ORDERS-MIN-INTEREST-RATE stops orders with an interest rate below 11 percent.
5. Making ORDER-ID-NUMBER the primary key causes the system to generate an index based on this column for quick access to the data records.

## CREATE TRIGGER/RULE

For an overview and examples of procedures and triggers, see Procedures and Triggers.

In the following table, YES indicates a valid execution method for this statement.

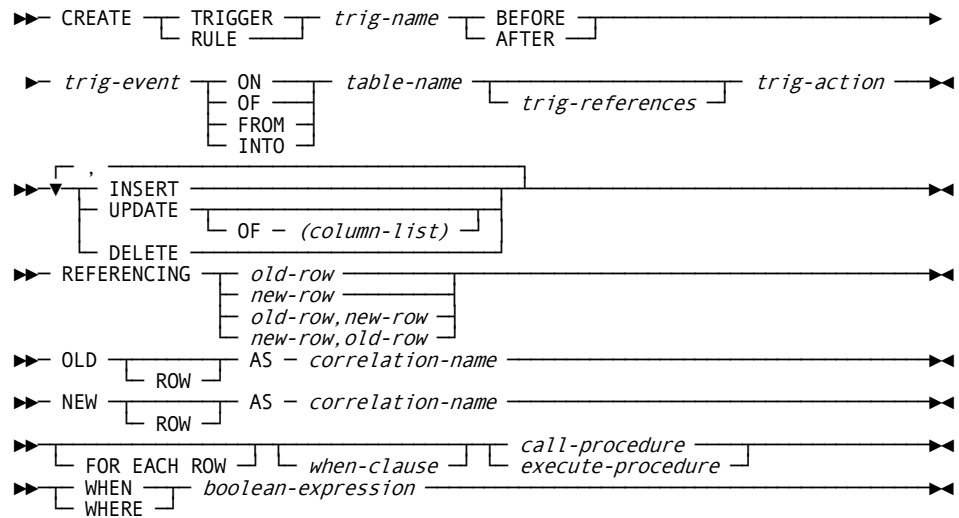
<b>This SQL statement can be executed in the following ways:</b>	<b>Through the CA Datacom Datadictionary Interactive SQL Service Facility (<i>interactive</i>)</b>	<b>In an application program prepared using a CA Datacom/DB SQL Preprocessor (<i>embedded</i>)</b>	<b>By using CA Dataquery (<i>SQL &amp; Batch Modes</i>)</b>
CREATE TRIGGER/RULE	YES	YES	YES

The CREATE TRIGGER/RULE statement creates a user-specified set of instructions whose execution is triggered by a specified maintenance request against a certain base table. See Transaction Integrity for information about when a created trigger is recognized by applications that are currently executing.

Note the following:

- Statement-level rules are not supported by CA Datacom/DB, that is, a TRIGGER/RULE can only be invoked for each row affected, not once for a statement that can affect many rows.
- Use of host variables, and therefore OUT and INOUT parameters, is forbidden in a procedure call that is part of a triggered action.
- To allow the trigger to be recognized during the execution of the plan, plans containing SQL statements that reference the table to which the trigger is being added are marked invalid and automatically rebound the next time they are executed.
- A single event can invoke multiple triggers. When this occurs, the triggers are invoked according to the date and time created with the most recently created being invoked last.

With regard to table partitioning, CREATE statements may not be issued against a table which is partitioned nor against a partition. For more information about table partitioning, see the *CA Datacom/DB Database and System Administration Guide*.



## CREATE TRIGGER/RULE

(Required) Specify for the CREATE either a TRIGGER or a RULE.

### **trig-name**

Specify the SQL-name of the trigger. SQL-names are SQL-identifiers.

### **BEFORE/AFTER**

Trigger BEFORE or AFTER the CA Datacom maintenance occurs.

### **trig-event ON/OF/FROM/INTO table-name**

Specify a trigger event, one of the listed options, and a table name. Options ON, OF, FROM, and INTO are supplied for syntax compatibility.

### **INSERT/UPDATE/DELETE**

Part of the *trig-event* syntax. You can use up to three comma-separated occurrences of these three maintenance types (INSERT, UPDATE, DELETE), but each maintenance type can only be used once in the *trig-event* definition.

### **OF**

(Optional) Part of the *trig-event* syntax. Supplied for syntax compatibility.

### **(column-list)**

is a list of columns whose value-change triggers a trigger. Omission implies all columns are triggers.

### **REFERENCING**

Part of the *trig-references* syntax.

**OLD**

Part of the old-row syntax. Refers to the row prior to UPDATE or DELETE.

**NEW**

Part of the new-row syntax. Refers to the row that results from execution of the INSERT or UPDATE.

**ROW**

*(Optional)* Part of the old-row and new-row syntax.

**AS correlation-name**

Part of the old-row and new-row syntax. The *correlation-name* is the name which is to be used to reference a column from the before-maintenance or after-maintenance image of the row being maintained. For example, if ACCOUNT\_NUMBER is a column name, OLD\_ACCOUNT\_ROW is the before-image *correlation-name*, and NEW\_ACCOUNT\_ROW, then OLD\_ACCOUNT\_ROW.ACCOUNT\_NUMBER can be used to refer to the account number prior to an UPDATE (or DELETE), and NEW\_ACCOUNT\_ROW.ACCOUNT\_NUMBER may be used to refer to the account number after an UPDATE (or INSERT) occurs. The use of this name is, of course, limited to the CREATE statement itself. Also see [Correlation Names](#) (see page 515) and [SQL Index Binding](#) (see page 516).

**FOR EACH ROW**

*(Optional)* Part of the *trig-action* syntax. Denotes this trigger as one that executes once per maintained row rather than once per SQL statement. ROW level is used (the default) even if you do not specify this optional clause.

**call-procedure**

Part of the *trig-action* syntax.

As a convenience to users of CREATE TRIGGER who need to pass large base table rows to a triggered procedure, a special DATACOM\_WHOLE\_ROW column is available. DATACOM\_WHOLE\_ROW is only visible during execution of the CREATE TRIGGER statement. When you pass the special DATACOM\_WHOLE\_ROW column name to your procedure, the procedure receives a CHAR parameter containing the entire row in CA Datacom/DB format. Following is an example of the use of DATACOM\_WHOLE\_ROW:

```
CREATE TRIGGER myTrigger AFTER DELETE FROM myTable
  REFERENCING OLD ROW AS old_row
  CALL handleMaintenance('DELETE', old_row.datacom_whole_row)
```

We recommend that you use this special DATACOM\_WHOLE\_ROW column name only if your parameter list becomes unwieldy because of an excessive number of columns being passed.



***execute-procedure***

Part of the *trig-action* syntax.

As a convenience to users of CREATE TRIGGER who need to pass large base table rows to a triggered procedure, a special DATACOM\_WHOLE\_ROW column is available. DATACOM\_WHOLE\_ROW is only visible during execution of the CREATE TRIGGER statement. See the information previously given in the *call-procedure* description about DATACOM\_WHOLE\_ROW.

**WHEN/WHERE *boolean-expression***

This is the when-clause syntax. The when-clause executes before the triggered action (the procedure call) is taken and cancels execution of the procedure if the *boolean-expression* evaluates to UNKNOWN or FALSE. The *boolean-expression* is similar to the WHERE clause of the SELECT statement but may not include subqueries or functions of any type. The *correlation-name* (see previous description) can be used to reference values in the row being maintained.

The trigger only activates on rows for which the *boolean-expression* evaluates to TRUE. If the WHEN clause containing the *boolean-expression* is omitted, the trigger always triggers.

## CREATE VIEW

In the following table, YES indicates a valid execution method for this statement.

<b>This SQL statement can be executed in the following ways:</b>	<b>Through the CA Datacom Datadictionary Interactive SQL Service Facility (<i>interactive</i>)</b>	<b>In an application program prepared using a CA Datacom/DB SQL Preprocessor (<i>embedded</i>)</b>	<b>By using CA Dataquery (SQL &amp; Batch Modes)</b>
CREATE VIEW	YES	YES	YES

To learn about using SQL keywords in CA Dataquery, see the *CA Dataquery User Guide*.

For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

The CREATE VIEW statement creates a view (derived table) of one or more tables or views.

When a CREATE VIEW statement successfully executes, a VIEW entity-occurrence is defined in CA Datacom Datadictionary in PROD status.

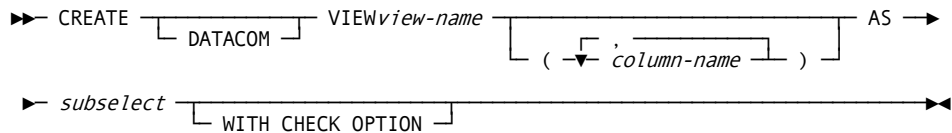
All of the tables and views specified in the CREATE VIEW statement must be in databases of the same security type, that is to say, either the CA Datacom/DB External Security Model or the SQL Security Model. See the *CA Datacom Security Reference Guide* for more information about security models.

## Privileges

If the Multi-User Facility is secured, whether the creator of the view may access that view depends on the security model of the view. Under the SQL Security Model, when a view is created the user who executes the CREATE VIEW statement is automatically granted all privileges on the view. Under the CA Datacom/DB External Security Model, all access rights, including those for the creator of a view, must be defined through the external security product. See the *CA Datacom Security Reference Guide* for more information.

With regard to table partitioning, CREATE statements may not be issued against a table which is partitioned nor against a partition. For more information about table partitioning, see the *CA Datacom/DB Database and System Administration Guide*.

Following is the syntax diagram for the CREATE VIEW statement:



## Description

### **DATAKOM**

*(Optional)* See DATAKOM VIEWS.

### ***view-name***

The unqualified or qualified name of the view. This name, including the implicit or explicit qualifier, must not name a table, view or synonym already described in the CA Datacom Datadictionary.

If you specify SQLMODE=ANSI or SQLMODE=FIPS in the Preprocessor options, the view name can be 1 to 18 characters in length.

If you specify SQLMODE=DATAKOM for extended mode in the Preprocessor options, the view name can be 1 to 32 characters in length.

The qualified form is the name preceded by an authorization ID and a period, for example, auth-id.view-name.

If you qualify the view-name, the qualifier designates the schema of the view. If you do not qualify the view-name, the default authorization ID is used as the qualifier.

If the CA Datacom/DB Security Facility is installed, the creator of the view is automatically granted the SELECT privilege on the view, and any other privilege that applies to a view and is a privilege that the creator has on the tables or views identified in the FROM clause of the SELECT statement. A privilege acquired by the creator is grantable only if the privilege from which it is derived is grantable by the creator.

### ***column-name***

A list of one or more names for columns in the view. The column names must be separated by commas and the list must be enclosed in parentheses.

If you specify SQLMODE=ANSI or SQLMODE=FIPS in the Preprocessor options, the column name can be 1 to 18 characters in length.

If you specify SQLMODE=DATAKOM for extended mode in the Preprocessor options, the column name can be 1 to 32 characters in length.

If you do not give other names to the columns, the columns of the view have the same names as the columns of the result table of the SELECT statement.

If the result of the SELECT statement has duplicate column names, or a column derived from a function, literal, or arithmetic expression, you must give names to all the columns. Give a name for each column and do not use the same name more than once.

**AS subselect**

The subselect defines the view. At any time, the view consists of the rows that result if the SELECT statement is executed.

The subselect must be the subselect form of a SELECT statement that does not reference host variables. See Subselect for the subselect syntax diagram and information about the subselect parameters.

**WITH CHECK OPTION**

Specifies that all inserts and updates against this view are checked to ensure that the newly inserted or updated row satisfies the view definition.

For example, if you use the view to insert a row, the row can be inserted only if you can also view it using this view, that is to say, the data being inserted must be within the bounds specified in the view's definition.

The WITH CHECK OPTION can be specified only if the view is updateable and the view definition does not include a nested subquery.

See [Example 3](#) (see page 710) for a view definition using the WITH CHECK OPTION.

## Processing

A view is read-only and cannot be updated if its definition includes any of the following phrases:

- A FROM clause naming more than one table or view (a join)
- The keyword DISTINCT
- A GROUP BY clause

Keywords and phrases which cannot be used in the definition of a view are:

- FOR UPDATE OF
- ORDER BY
- UNION

**Note:** A view whose definition involves either a GROUP BY clause or a column function cannot be named in any FROM clause that contains another table or view, that is to say, the view cannot be joined. This is not a restriction on the content of the CREATE VIEW statement itself, but on queries that reference the view. An error message is issued if you violate this rule. Also, a view whose definition contains a SELECT DISTINCT may only be joined if the join itself has a SELECT DISTINCT. Again, this is a restriction on queries that reference the view, not a restriction on the view itself.

## Example 1

Create a read-only view with the specifications listed in the following:

1. View name: PROJV1
2. Column names: PROJNO, PROJNAME, PROJDEP, RESPEMP, EMPNO, FIRSTNME, MIDINIT, LASTNAME
3. The view joins the tables PROJTBL and NAMETBL where a value in the RESPEMP column is equal to a value in the EMPNO column.

**Note:** Since this view joins two tables, it is a read-only view.

```
EXEC SQL
    CREATE VIEW PROJV1
        (PROJNO, PROJNAME, PROJDEP, RESPEMP,
         FIRSTNME, MIDINIT, LASTNAME)
    AS SELECT ALL
        PROJNO, PROJNAME, DEPTNO, EMPNO,
        FIRSTNME, MIDINIT, LASTNAME
    FROM PROJTBL, NAMETBL
    WHERE RESPEMP = EMPNO
END-EXEC
```

## Example 2

Create an updateable view with the specifications listed in the following:

1. View name: SUPPLY
2. Column names: SNUM, SNAME, STATUS, CITY
3. Search condition: The value of STATUS must be greater than 10

This view is a row and column subset view of the table CASUPL which contains columns SNUM, SNAME, STATUS, CITY, STATE and ZIP. View SUPPLY can be used to update values for the four specified columns. However, this view would not be used to insert new rows into CASUPL since some of the columns are not included in the view definition.

```
EXEC SQL
    CREATE VIEW SUPPLY
        (SNUM, SNAME, STATUS, CITY)
    AS SELECT ALL
        SNUM, SNAME, STATUS, CITY
    FROM CASUPL
    WHERE STATUS > 10
END-EXEC
```

### Example 3

Create an updateable view with the specifications listed in the following:

1. View name: SUPPLY
2. Column names: SNUM, SNAME, STATUS, CITY, STATE, ZIP
3. Search condition: The value of STATUS must be greater than 10
4. Use the WITH CHECK OPTION

This view is a row subset view of the table CASUPL which contains columns SNUM, SNAME, STATUS, CITY, STATE and ZIP. View SUPPLY can be used to update values for all the columns, and to insert new rows into CASUPL where the value for STATUS is greater than 10. If you attempt to insert a row where the value of STATUS is less than or equal to 10, the row is not inserted because the WITH CHECK OPTION does not allow you to view any row where STATUS is not greater than 10.

```
EXEC SQL
    CREATE VIEW SUPPLY
        (SNUM, SNAME, STATUS, CITY, STATE, ZIP)
    AS SELECT ALL
        SNUM, SNAME, STATUS, CITY, STATE, ZIP
    FROM CASUPL
    WHERE STATUS > 10
    WITH CHECK OPTION
END-EXEC
```

## DECLARE CURSOR

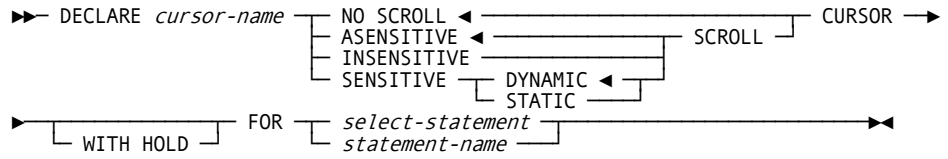
This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
DECLARE CURSOR		YES	

**Note:** YES indicates a valid execution method for this statement.

**Note:** This statement cannot be executed interactively. Use the SELECT statement instead (see [SELECT](#) (see page 766)). For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

The DECLARE CURSOR statement defines a cursor.

Following is the syntax diagram for the DECLARE CURSOR statement:



## Description

### **NO SCROLL**

Specifies the cursor is not scrollable. If scrolling is not specified, the default is NO SCROLL. With NO SCROLL, the FETCH statement can only return the next row.

**Note:** Rows are sensitive to changes if a temporary table is not used.

### **SCROLL**

Specifies the cursor is scrollable.

### **ASENSITIVE**

With ASENSITIVE specified, the cursor is SENSITIVE DYNAMIC if a temporary table is not used. Otherwise, it is INSENSITIVE. This is the same as NO SCROLL except that scroll options are available with the FETCH statement.

When scrolling is used, ASENSITIVE is the default scroll type.

### **INSENSITIVE**

INSENSITIVE means that the cursor is not sensitive to changes in the underlying tables of the result set.

With INSENSITIVE specified, the cursor is made read-only, and no positioned update or delete is allowed

### **SENSITIVE**

SENSITIVE means that the cursor is sensitive to changes made after the cursor is opened.

**Note:** If changes cannot be made visible to the cursor, an error is returned on the bind of the cursor open statement.

Changes cannot be made visible if a temporary table is required, or when the cursor has more than one database table, or when a User Defined Function Table (UDFT) is used.

A SENSITIVE scroll cursor must therefore reference only a single database table and not use aggregation.

SENSITIVE scroll cursors can be DYNAMIC or STATIC. The DYNAMIC scroll cursor is the default with SENSITIVE specified.

### **DYNAMIC**

DYNAMIC means that changes made by the current transaction are visible immediately, and changes made by other transactions are visible when committed.

A temporary table is not built, so qualifying rows that are inserted while the cursor is open are visible, and rows deleted or updated such that they no longer qualify while the cursor is open are no longer visible.



If a row is updated, it is logically moved to its new position in the result set. For example, if the key being used to retrieve rows dynamically is on column `line_number`, if the application increments `line_number` using a positioned update, then the same row will be fetched using `fetch next`.

This is more efficient than an insensitive scroll cursor because no temporary table is required, but the application must be able to work properly with the dynamic nature of the result set.

### **STATIC**

As with an insensitive scroll cursor, a result table is built, so the size of the result set does not change. However, rows fetched using the `fetch sensitive` option reflect the current state of the corresponding underlying base table row.

Row of a cursor declared as sensitive static can be fetched as sensitive or insensitive. If `fetch sensitive` is used, then the temporary table is updated to reflect the corresponding row of the underlying database table. If `fetch insensitive` is used, the row is returned from the temporary table in its current state, which reflects changes due to a previous `fetch sensitive`.

- Rows inserted after the cursor is opened are not visible.
- Rows deleted after the cursor is opened are not visible; a delete hole is created.
- Rows updated after the cursor is opened such that they no longer qualify are not visible; an update hole is created.

### ***cursor-name***

A cursor with the specified name is defined when your program is executed. The name must not be the same as the name of another cursor declared in your program.

If you specify `SQLMODE=ANSI` or `SQLMODE=FIPS` in the SQL Preprocessor options, the cursor name can be 1 to 18 characters.

For all other modes, the cursor name can be 1 to 32 characters.

### **WITH HOLD**

When this CA Datacom/DB extension is specified, the cursor stays open when a `COMMIT WORK` is executed. Any record-at-a-time command that commits the logical unit of work (for example `LOGCP`, `LOGCR`) works the same way. This is especially useful in a batch program that does updates and issues `COMMIT WORK` periodically to keep the log from becoming full and to limit the amount of work `RESTART` would have to do in case of a failure. The repositioning of the cursor is harder to program without `WITH HOLD`.

### ***select-statement***

For information about the select-statement, see [Select-Statement](#).

### ***statement-name***

Specifies the name of a prepared statement. That statement must be prepared (using the PREPARE statement) sometime after the DECLARE CURSOR statement is executed and before the OPEN statement is executed, and it must be a select-statement. For information on the PREPARE statement, see [PREPARE](#) (see page 756).

A cursor in the open state designates a result table and a position relative to the rows of that table. The table is the result table specified by the select-statement of the cursor.

The result table is read-only if the select-statement includes any of the following:

- The keyword DISTINCT
- A UNION operator
- A column function
- A GROUP BY clause
- A HAVING clause
- An ORDER BY clause

The result table is also read-only if the FROM clause of the outer subselect of the select-statement:

- Identifies more than one table or view
- Identifies a read-only view
- Identifies a table or view that is also identified in a FROM clause of a subquery of the select-statement

## Cursor Usage

A DECLARE CURSOR statement must have corresponding OPEN, FETCH and CLOSE statements in the same source program for the execution of the OPEN to proceed without error.

Cursors must be declared in the source:

1. Before any reference to the cursor, and
2. After all host variables used in the definition have been defined.

If an exception declaration (WHENEVER statement) is not provided, the recommended practice is for your program to include code to check the returned value of the SQLCODE immediately after each executable SQL statement. If you do not use a WHENEVER statement, however, be aware of the following:

Cursor definitions are declarations, not operational (procedural) statements, and as such are used for Preprocessor input only. Because no call is sent to the Multi-User Facility, checking the SQLCODE after a DECLARE CURSOR statement always shows the SQLCODE received by whatever statement immediately preceded the DECLARE CURSOR statement.

In COBOL, cursor declarations can be made in the:

- WORKING-STORAGE SECTION
- LINKAGE SECTION
- REPORT SECTION
- PROCEDURE DIVISION

**Note:** The recommended practice is to place all cursor definitions immediately before the PROCEDURE DIVISION source statement. But if you do not use a WHENEVER statement and want to avoid any potential confusion resulting from the content of the SQLCODE after a DECLARE CURSOR statement, place all of your DECLARE CURSOR statements in the WORKING-STORAGE SECTION.

## Example1

In this example, the DECLARE CURSOR statement associates the cursor name C1 with the results of the SELECT. This example also shows how the cursor is opened with the OPEN statement, used in a FETCH statement, and closed with a CLOSE statement.

```
EXEC SQL
  DECLARE C1 CURSOR FOR
    SELECT DEPTNO, DEPTNAME, MGRNO
    FROM DEPTTBL
    WHERE ADMDEPT = 'A0'
END-EXEC.

...
* DISPLAY DEPT TABLE INFO
EXEC SQL
  WHENEVER NOT FOUND   GO TO DEPT-FETCH-LP-END
  WHENEVER SQLERROR    GO TO SQL-ERROR-RTN
  WHENEVER SQLWARNING  CONTINUE
END-EXEC.

EXEC SQL
  OPEN C1
END-EXEC.
```

```

DEPT-FETCH-LP.
  EXEC SQL
    FETCH C1 INTO :DNUM, :DNAME, :MNUM
  END-EXEC.
  DISPLAY DNUM, DNAME, MNUM.
  GO TO FETCH-LOOP.
DEPT-FETCH-LP-END.

  EXEC SQL
    CLOSE C1
  END-EXEC.
...

```

## Example 2

In this example, the DECLARE CURSOR statement defines the cursor, C1. This cursor could be used in an UPDATE statement to update the column named in the clause, or used in a DELETE statement to delete a row.

```

EXEC SQL
  DECLARE C1 CURSOR FOR
    SELECT DEPTNO, DEPTNAME, MGRNO
    FROM DEPTTBL
    WHERE EXISTS
      (SELECT *
       FROM DIVTBL
       WHERE DIVTBL.DEPTNO = DEPTTBL.DEPTNO)
END-EXEC

```

## Example 3

In this example, the DECLARE CURSOR statement defines the cursor, C1, and orders the results of the SELECT since an ORDER BY clause is included in the definition. The SELECT retrieves the number and name of all employees hired before 1980 in order of seniority. HIREDATE is in the form "yymmdd."

```

EXEC SQL
  DECLARE C1 CURSOR FOR
    SELECT EMPNO, LASTNAME, FIRSTNAME, HIREDATE
    FROM EMP
    WHERE HIREDATE < 800000
    ORDER BY HIREDATE
END-EXEC

```

## DECLARE STATEMENT

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
DECLARE STATEMENT		YES	

**Note:** YES indicates a valid execution method for this statement.

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

Following is the syntax diagram for the DECLARE statement (statement-name specifies the 1- to 18-character(s) name of a prepared SQL statement):

➤ DECLARE *statement-name* STATEMENT ➤

## DELETE

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
DELETE (positioned)		YES	
DELETE (searched)	YES	YES	YES

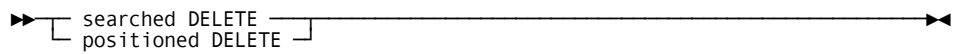
**Note:** YES indicates a valid execution method for this statement. To learn about using SQL keywords in CA Dataquery, see the *CA Dataquery User Guide*. For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

The DELETE statement deletes rows from a table or view. Deleting a row from a view deletes the row from the table that the view is based on.

You must specify ISOLEVEL=C (isolation level C) in the Preprocessor options when using the DELETE statement

**Note:** When the NOMAINT option of the CA Datacom/DB Utility (DBUTLTY) ACCESS function is in force, a DELETE statement receives a CA Datacom/DB return code 94(87), where 87 is a decimal internal return code (hex 57) that tells you no maintenance statements are allowed while NOMAINT is in force.

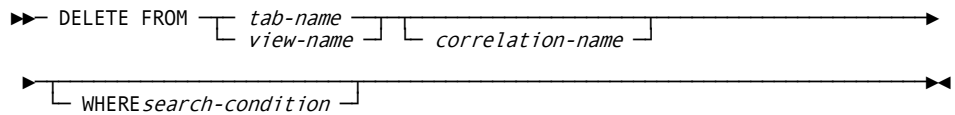
Following is the syntax diagram for a DELETE statement:



**Note:** The positioned DELETE is not used by CA Dataquery.

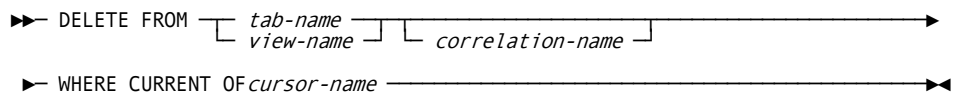
## Searched DELETE

Following is the syntax diagram for the searched DELETE statement:



## Positioned DELETE

Following is the syntax diagram for the positioned DELETE statement, which you use with a cursor.



---

## Description

**FROM *tab-name* or FROM *view-name***

Name the table or view from which you want to delete. The table or view must be described in the CA Datacom Datadictionary, but must not be a CA Datacom Datadictionary table or a read-only view.

***correlation-name***

You can specify a 1- to 18-byte correlation-name to be used within the search-condition to designate the table or view. Also see Correlation Names and SQL Index Binding.

**WHERE**

Introduces a condition that specifies what rows are to be deleted. You can omit the clause, give a search condition or name a cursor. If you omit the clause, all rows of the table or view are deleted.

***search-condition***

Each column-name in the search condition must name a column of the table or view, and the table or view must not be referenced in the FROM clause of any subselect in the search condition. See Search Conditions for the search-condition syntax diagram.

Each column-name in the search condition which is not in a subquery must name a column of the table or view being updated. In subqueries, the table or view being updated must not be named in any FROM clause.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a row. The result of the subquery is used in applying the search condition. In actuality, the subquery is executed for each row only if it contains a correlated reference to a column of the table or view.

**CURRENT OF *cursor-name***

Names a cursor that is defined in a DECLARE CURSOR statement of your program. The DECLARE CURSOR statement must appear in your program before the DELETE statement.

The table or view named must also be named in the FROM clause of the SELECT statement of the cursor. The result table of the cursor must not be read-only.

When the DELETE statement is executed, the cursor must be positioned on a row. This row is the one that is deleted. After the deletion, the cursor is positioned before the next row of its result table. If there is no next row, the cursor is positioned after the last row.

### Processing

No rows are deleted if an error occurs during the execution of a DELETE statement. The cursor is closed if an error occurs that makes the position of a cursor unpredictable.

Unless appropriate locks already exist, one or more exclusive locks are acquired by the execution of a successful DELETE statement. Unless the locks are released by a rollback operation, a deleted row cannot be accessed by any unit of recovery.

### Example 1

Delete the row from table DEPTTBL where the value of DEPTNO equals D1.

```
EXEC SQL  
  
DELETE FROM DEPTTBL  
WHERE DEPTNO = 'D1'  
END-EXEC
```

### Example 2

Delete several rows from table NAMETBL, specifically, those for all employees in Department E1 or D2.

```
EXEC SQL  
    DELETE FROM NAMETBL  
    WHERE WORKDEPT = 'E1' OR WORKDEPT = 'D2'  
END-EXEC
```



### Example 3

Deletes the row where the cursor is positioned.

```
EXEC SQL
  DECLARE C1 CURSOR FOR
    SELECT DEPTNO, DEPTNAME, MGRNO
    FROM DEPTTBL
    WHERE ADMDEPT='A'

END-EXEC.

EXEC SQL
  OPEN C1
END-EXEC.

FETCH-LOOP.
  IF SQLCODE =
    EXEC SQL
      FETCH C1 INTO :DNUM, :DNAME, :MNUM
    END EXEC.
    EXEC SQL
      DELETE FROM DEPTTBL
      WHERE CURRENT OF C1
    END-EXEC.
    GO TO FETCH LOOP.

EXEC SQL
  CLOSE C1

END-EXEC.
```

## DESCRIBE

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
DESCRIBE		YES	

**Note:** YES indicates a valid execution method for this statement.

The DESCRIBE statement obtains information about a specified table or view, or about a prepared statement.

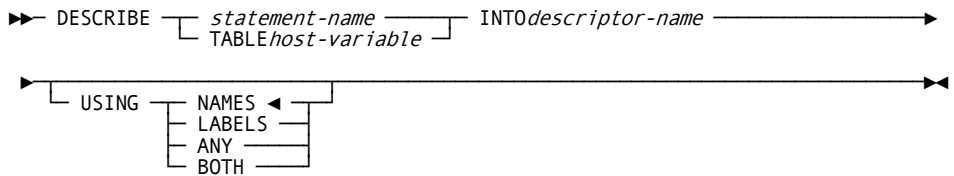
**Note:** When you use the PREPARE statement to create an executable SQL statement from a character string form of the statement, the executable form is called a prepared statement. Information about a prepared statement can also be obtained by using the INTO clause of the PREPARE statement. See [PREPARE](#) (see page 756) for more information about PREPARE.

No authorization is required to DESCRIBE a prepared statement, but to execute a DESCRIBE TABLE statement you must hold at least one of the following authorizations:

- INSERT, UPDATE, DELETE, SELECT, INDEX, or ALTER on the table or view.
- Be a global, database, or table owner.
- Be the original creator of the table (only applies if the SQL Security Model is being used).

**Note:** Because the table name is not known at prepare time, security checks for this statement are always performed at execution time regardless of plan security options.

Following is the syntax diagram for the DESCRIBE statement:



---

## Description

***statement-name***

Identifies the prepared statement about which you want to obtain information.

**TABLE *host-variable***

Identifies the table or view. When the DESCRIBE statement is executed, *host-variable* must be a character string data type (a CHAR or VARCHAR variable) that contains a name identifying a table or view. If the escape character is used in the string it must be the quotation mark (double-quotes). *Host-variable* must be preceded by a colon. An indicator variable is not allowed.

**INTO *descriptor-name***

Identifies an SQL Descriptor Area (SQLDA) to be filled. See SQLDA (DESCRIBE or PREPARE INTO Statements). If the TABLE clause is used to describe a table or view, the information returned in the SQLDA describes the columns of the specified table or view. If *statement-name* is used to describe a select-statement associated with a dynamic cursor, the information returned in the SQLDA describes the columns of the result table of the select-statement. When any other prepared statement is described, the SQLD field of the SQLDA is set to zero, which indicates a statement other than a select-statement has been described.

**USING**

Specifies if the SQLNAME field in the SQLDA is to contain a column name or a column label. If the requested value does not exist, SQLNAME is set to length 0.

**NAMES**

Assigns the name of the column. This is the default.

**LABELS**

Assigns the label for the column. A label is the column's CA Datacom Datadictionary field attribute HEADING-1.

**ANY**

Assigns the column label, or (if one does not exist) the column name.

**BOTH**

Assigns the column names to the first *n* occurrences of SQLVAR, and the column labels to the second *n* occurrences.

## Example

In the following COBOL example of an SQLDA, a table or view with up to 100 columns is described. (For more information on SQLDAs used in DESCRIBE statements, see SQLDA (DESCRIBE or PREPARE INTO Statements).)

```
01  SQLDA
   05  SQLAID                PIC X(8)  VALUE 'SQLDA  '.
   05  SQLABC                PIC S9(9)  COMP.
   05  SQLN                  PIC S9(4)  COMP VALUE +100.
   05  SQLD                  PIC S9(4)  COMP.
   05  SQLVAR OCCURS 100 TIMES.
       10  SQLTYPE           PIC S9(4)  COMP.
       10  SQLLEN            PIC S9(4)  COMP.
       10  FILLER REDEFINES SQLLEN.
           15  SQLPREC       PIC X.
           15  SQLSCALE     PIC X.
       10  SQLDATA           PIC S9(9)  COMP.
       10  SQLIND            PIC S9(9)  COMP.
       10  SQLNAME-VARCHAR.
           49  SQLNAME-LEN   PIC S9(4)  COMP.
           49  SQLNAME      PIC X(30)  .

01  TABLE-NAME              PIC X(32)  .

MOVE table-name TO TABLE-NAME.
EXEC SQL
DESCRIBE TABLE :TABLE-NAME INTO :SQLDA USING ANY
END-EXEC
PERFORM PRINT-NAMES SQLD TIMES.
```

See the previous example and consider the following.

Before the DESCRIBE statement is executed, you must set SQLN to the number of SQLVAR occurrences that are provided in the SQLDA. No information is returned in the SQLVARs if SQLD is set greater than SQLN, because in this case the SQLDA is not large enough (see the following for an explanation of how the SQLD is set).

CA Datacom/DB sets:

- SQLAID to SQLDA.
- SQLABC to the length of the SQLDA.
- SQLD to the number of columns described by occurrences of SQLVAR (or twice this number when USING BOTH is specified).

Assuming the object being described has n columns (and there are enough SQLVAR entries), values are assigned to the first n SQLVAR fields (listed in the following) to describe the object's columns. (When USING BOTH is specified, the SQLNAME field is used in a second set of n SQLVAR entries to return labels.)

**SQLTYPE**

A code for the data type of the column and if the column is nullable. See [SQLDA \(DESCRIBE or PREPARE INTO Statements\)](#) (see page 870) for a list of these codes.

**SQLLEN**

A length value depending on the data type of the column. See [SQLDA \(DESCRIBE or PREPARE INTO Statements\)](#) (see page 870) for possible values.

**SQLDATA**

A value of -1 indicates FOR BIT DATA.

**SQLIND**

Reserved.

**SQLNAME**

The unqualified name or label of the column, depending on the USING option specified.

**Note:** A string length of zero indicates that the column name or label does not exist.

## DROP

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
DROP INDEX	YES	YES	
DROP SYNONYM	YES	YES	YES
DROP TABLE	YES	YES	YES
DROP VIEW	YES	YES	YES
DROP PROCEDURE	YES	YES	YES
DROP TRIGGER/RULE	YES	YES	YES

**Important!** There is also a DROP PLAN statement that can only be submitted through DBSQLPR. See DROP PLAN (DBSQLPR).

To learn about using SQL keywords in CA Dataquery, see the *CA Dataquery User Guide*.

For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

The DROP statement is a CA Datacom/DB extension. DROP removes an SQL-accessible object. The object's description is removed from the CA Datacom Datadictionary. Any application plans that reference the object are invalidated.

**Important!** When an object is dropped, any objects that are directly or indirectly dependent on that object are either dropped or marked for rebind.

When a table or view is dropped, the prepared statements in plans that reference the table or view are marked nonexecutable until rebound. An attempt to execute these statements invokes automatic rebind, and the rebind fails. If the plans containing referencing statements are currently executing or binding, the DROP request is aborted. This is always the case when the plan containing the DROP statement references the table or view, because this is an attempt to invalidate its own plan and is not allowed. Dropping an index causes all plans dependent on the indexed table to be 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 on Relationship Reports.

The DROP statement does not process and you receive a -118 return code when the CA Datacom Datadictionary entity-occurrence definition of the table, view, synonym, or index you specify is protected with a password or a Lock Level 1 or 2. The error message also includes a Datadictionary Service Facility (DSF) return code. The DSF return codes are:

- IPW (for password protected)
- IOR (for Lock Level 1 protected)
- NTF (for Lock Level 2 protected)

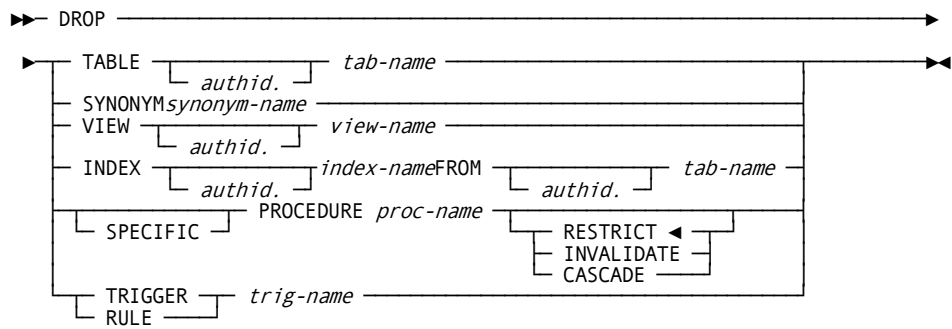
See Deleting SQL Objects for more information. See [Preliminary Information—Lock Levels](#) (see page 597) and to Datadictionary documentation for more information on passwords and lock levels.

**Note:** When the NOMAINT option of the CA Datacom/DB Utility (DBUTLTY) ACCESS function is in force, a DROP statement receives a CA Datacom/DB return code 94(87), where 87 is a decimal internal return code (hex 57) that tells you no maintenance statements are allowed while NOMAINT is in force.

Also note that the DROP statement is rejected if other DML statements (see [SQL Statements](#) (see page 597)) are in the same plan or if the table being dropped is in use by other plans.

With regard to table partitioning, DROP statements may not be issued against a table which is partitioned nor against a partition. For more information about table partitioning, see the *CA Datacom/DB Database and System Administration Guide*.

Following is the syntax diagram for the DROP statement:



**Note:** DROP INDEX is not used by CA Dataquery.

## Description

***authid.***

An optional identifier of the schema for the table, view, or index. If specified for the index-name, it must be the same as the authorization ID of the table you have specified with the tab-name in the FROM tab-name clause. Use a period (.) to concatenate the authorization ID to the table or view name, for example, authid.tab-name.

**TABLE**

Indicates that the object you are dropping is a table.

***tab-name***

Specify the name of the table you want to drop. The tab-name must identify a table described in the CA Datacom Datadictionary, other than a CA Datacom Datadictionary table.

The DROP TABLE statement removes the table from the schema, obsoletes the table from the CA Datacom Datadictionary, and removes the table from the CA Datacom/DB Directory (CXX).

**Important!** The table data is deleted and the space is reclaimed when you drop a table.

All views and synonyms based on the table are dropped. All application plans and statements that reference the table are invalidated. If the table definition exists in TEST or HIST status in CA Datacom Datadictionary, those status/versions are deleted at the same time as the PROD status.

All columns (columns appear as FIELD occurrences in CA Datacom Datadictionary ), keys, elements, and support data related to the table are also obsoleted. The support data includes aliases, descriptors, relationship definitions, and text (SQL comments).

**SYNONYM**

Indicates that the object you are dropping is a synonym.

***synonym-name***

Specify the name of the synonym you want to drop. Dropping a synonym does not affect the table or view referenced by the dropped synonym.



**VIEW**

Indicates that the object you are dropping is a view.

***view-name***

Specify the name of the view you want to drop. The view name must identify a view described in the CA Datacom Datadictionary. The definition of the view is removed from the CA Datacom Datadictionary. The definition of any view that is directly or indirectly dependent on that view is also removed. When the definition of a view is removed from the CA Datacom Datadictionary, all privileges on that view are also dropped.

**INDEX**

Indicates that the object you are dropping is an index.

***index-name FROM tab-name***

Specify the name of the index you want to drop and the table to which the index belongs. Make certain that the index-name you use in the DROP INDEX statement matches the SQLNAME attribute of the KEY that you wish to DROP.

Dropping an index removes the index from the Index Area (IXX), removes the index definition from the Directory (CXX) and CA Datacom Datadictionary, and causes all plans dependent on the indexed table to be 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 on Relationship Reports.

**SPECIFIC**

Provided for ANSI syntax compatibility only. Specifies that the proc-name given is the *specific name* of the procedure.

**PROCEDURE**

DROP PROCEDURE deletes a procedure definition and the associated program definition and plan (that is, the portion of the procedure logic that resides in SQL).

**Note:** For an overview and examples of procedures and triggers, see [Procedures and Triggers](#) (see page 70).

***proc-name***

Specify a procedure name.

ANSI supports *overloading* of the procedure name, with duplicate procedure names resolved to procedure definitions using the layout of the parameter list. For syntax compatibility purposes, a second, *specific* name may be given to a procedure to uniquely identify it, but it must match the nonspecific name, and the nonspecific name must be unique.

**RESTRICT**

Tells SQL to abort the DROP if triggers exist that call the procedure. RESTRICT is the default.

**INVALIDATE**

Specifies that referencing triggers are marked invalid, and therefore cannot execute until the procedure is re-added and the triggers rebound. The rebound occurs automatically and is transparent to the user unless the procedure has not been replaced when the triggering of the trigger occurs. This option protects the integrity of the tables on which the triggers are defined. Any attempt to fire an invalid trigger causes the triggering maintenance request (INSERT, UPDATE, or DELETE) to abort, unless the automatic trigger rebound succeeds. The INVALIDATE option also causes the plan related to the procedure to be marked invalid. The plan automatically rebounds during the next CALL PROCEDURE if the procedure is re-created.

**CASCADE**

Specifies the dropping of any referencing triggers. In order to avoid the potential for later PROGRAM/PLAN MISMATCH errors if the procedure is re-precompiled, unless the program is called by other procedures we strongly recommend that you delete the corresponding load module and CA Datacom Datadictionary PROGRAM definitions at this time. The LOAD module and CA Datacom Datadictionary PROGRAM definitions are not affected.

**DROP TRIGGER/RULE**

Specify either a TRIGGER or RULE to be dropped. DROP TRIGGER/RULE deletes a trigger or rule from the system. To prevent a trigger from being invoked after being dropped, plans containing SQL statements that reference the table from which the trigger is being removed are marked invalid and automatically rebound the next time they are executed.

**Note:** For an overview and examples of procedures and triggers, see [Procedures and Triggers](#). (see page 70)

***trig-name***

Specify a trigger name.

## Example 1

Drop table TED.TDEPT. TED is the authorization ID of the user who owns the table.

```
EXEC SQL
      DROP TABLE TED.TDEPT
END-EXEC
```

## Example 2

Drop the view VDEPT.

```
EXEC SQL
      DROP VIEW VDEPT
END-EXEC
```

## Example 3

Drop the index named EMPLOYEE\_INDEX that belongs to the table named EMPLOYEES.

```
EXEC SQL
      DROP INDEX EMPLOYEE_INDEX FROM EMPLOYEES
END-EXEC
```

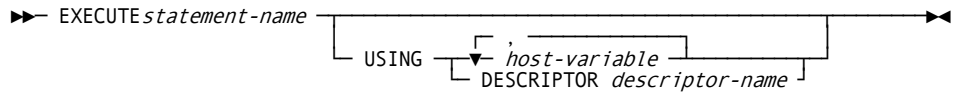
## EXECUTE

This SQL statement can be executed in the following ways:	Through the Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
EXECUTE		YES	

**Note:** YES indicates a valid execution method for this statement.

No privileges are required to execute an EXECUTE statement, and no authorization is required to preprocess an EXECUTE statement. That is, there is no security check of the EXECUTE statement or its contents at preprocessor time. At execution time, however, the statement's contents *are* checked against the security conditions that are valid for that program.

Following is the syntax diagram for the EXECUTE statement:



## Description

### **statement-name**

Identifies the prepared statement to be executed. The prepared statement must have been prepared within the same logical unit of work as the EXECUTE statement used to execute it, and it cannot be a select-statement. (Prepared select-statements are executed with OPEN statements.) See [PREPARE](#) (see page 756) for more information about prepared statements.

### **USING**

If the prepared statement includes parameter markers, you must code the USING clause. The USING clause specifies either a list of host variables (whose values are substituted for parameter markers) or a SQL Descriptor Area (SQLDA) that contains a description of the host variables. If you code the USING clause when there are no parameter markers in the prepared statement, the USING clause is ignored. See [Rules for Parameter Markers](#) (see page 758) for rules for parameter markers. Information about parameter marker replacement can be found on [PREPARE](#) (see page 756).

### **host-variable**

Identifies host structures or variables used to supply the values for parameter markers. Separate the host variables in the list with commas.

The host-variable specified in the USING clause identifies a structure or variable that is described in the program in accordance with the rules for declaring host structures and variables. When the statement is executed, a reference to a structure has been replaced by references to each of its variables. The number of variables must be the same as the number of parameter markers in the prepared statement. The *n*th variable supplies the value for the *n*th parameter marker in the prepared statement.

**DESCRIPTOR *descriptor-name***

The descriptor-name identifies a SQL Descriptor Area (SQLDA) containing a description of the host variables. The SQLD field (used to indicate the number of variables) must be set to the number of parameter markers in the prepared statement, and the length of the SQLDA (indicated by SQLABC) must be sufficient to describe that number of variables. The nth variable (the nth SQLVAR entry) described by the SQLDA corresponds to the nth parameter marker in the prepared statement. For details on the SQLDA, see [SQL Descriptor Area \(SQLDA\)](#) (see page 869)

## Parameter Marker Replacement

Before the prepared statement is executed, host variables are assigned to target variables for each parameter marker. The attributes of the target variables depend on the role that the parameter marker plays in the SQL statement. The rules for these roles are shown in the following. In these rules, P represents the parameter marker in question.

***Arithmetic Operand***

When P is an operand of an infix operator, the data type, precision, and scale of the target variable is the same as the other operand, so both operands cannot be parameter markers.

When P is the operand of a unary minus, the data type of the target variable is double precision floating-point.

***LIKE Predicate***

If you specify the parameter marker as the pattern of a LIKE predicate (see [LIKE Predicate](#) (see page 585)), ensure that the host variable used to replace the parameter marker is compatible with the first operand.

***Comparand***

P can be a comparand in a BETWEEN, IN, or basic predicate, where at least one of the other comparands is not a parameter marker. The attributes of the target for P are those of one of the other comparands.

- For a BETWEEN predicate, it is the first comparand that is a column name, or (if no comparand is a column name) the first comparand that is not a parameter marker.

- For the IN predicate, it is the first comparand that is not a parameter marker.
- For a basic predicate, it is the other comparand.

#### Assignment Operand

P must be the value assigned to a column in an INSERT or UPDATE statement. The attributes of the target are the same as the column.

#### Example

```

01 S1.
   49 S1LEN          PIC S9999 COMP VALUE +80.
   49 S1HV           PIC X(8).
01 HV1              PIC X VALUE 'X'.
01 HV2              PIC X VALUE 'Y'.

MOVE 'INSERT INTO T1 VALUES (?, ?)' TO S1HV.
EXEC SQL
    PREPARE STMT1 FROM S1
END-EXEC
EXEC SQL
    EXECUTE STMT1 USING :HV1, :HV2
END-EXEC

```

## EXECUTE IMMEDIATE

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
EXECUTE IMMEDIATE		YES	

**Note:** YES indicates a valid execution method for this statement.

The EXECUTE IMMEDIATE statement prepares and executes an SQL statement. EXECUTE IMMEDIATE is more convenient than using PREPARE and EXECUTE, but note the following:

- With EXECUTE IMMEDIATE, parameter markers cannot be used, and
- EXECUTE IMMEDIATE is less efficient than using PREPARE and EXECUTE, which allow you to prepare a statement once and execute it multiple times.

No authorization is required to preprocess an EXECUTE IMMEDIATE statement, that is, there is no security check of the EXECUTE IMMEDIATE statement or its contents at preprocessor time. At execution time, however, the statement's contents *are* checked against the security conditions that are valid for that program.

Following is the syntax diagram for the EXECUTE IMMEDIATE statement:

```

▶▶ EXECUTE IMMEDIATE ┌── string-expression ──┐──────────────────────────────────▶
                    │ host-variable         │
  
```

## Description

### **string-expression**

A PL/I expression that yields a character string.

### **host-variable**

A host-variable must be used for languages other than PL/I. It must identify a varying-length string host variable.

## Rules for Statement Strings

A statement string is the value of the string-expression or host-variable. See the rules for statement strings on [Rules for Statement Strings](#) (see page 758).

## Example

```

01 S1.
   49 S1LEN          PIC S9999 COMP VALUE +80.
   49 S1HV           PIC X(8) .

MOVE 'INSERT INTO T1 VALUES (1, 2)' TO S1HV.
EXEC SQL
      EXECUTE IMMEDIATE S1
END-EXEC
  
```

## EXECUTE PROCEDURE

For information about the EXECUTE PROCEDURE statement, see [CALL/EXECUTE PROCEDURE](#) (see page 610).

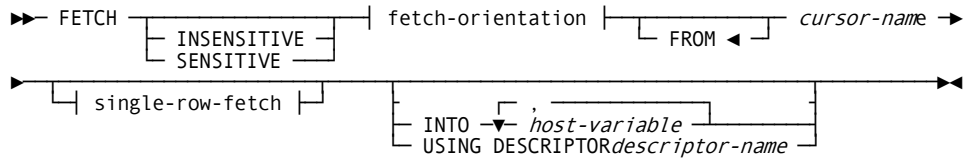
# FETCH

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
FETCH		YES	

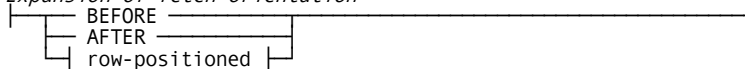
**Note:** YES indicates a valid execution method for this statement. For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

The FETCH statement positions a cursor on the specified row of its result table and assigns the value of that row to any specified host variables. You can use multiple FETCH statements referencing the same cursor. The host variables in the INTO list are matched by position to SELECT list expressions. The INTO list may FETCH a leading subset of the SELECT list expressions. However, unless the plan has SQLMODE=DB2, the FETCH statement that appears first in the host program must have the same or greater number of host variables. Also, although each FETCH statement may reference different host variables, unless the plan has SQLMODE=DB2, they must have the same data type, length, precision, and scale.

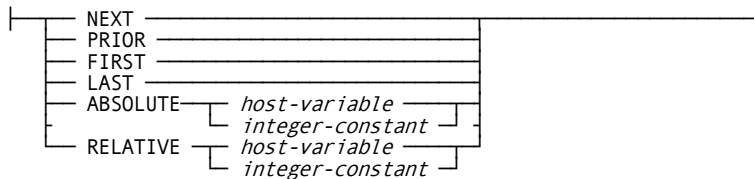
Following is the syntax diagram for the FETCH statement:



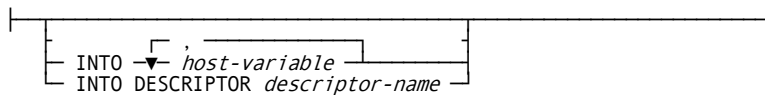
Expansion of fetch-orientation



Expansion of row-positioned



Expansion of single-row-fetch





**Note 1:** Whether INSENSITIVE or SENSITIVE is the default in the FETCH statement depends on the sensitivity of the cursor specified in the DECLARE CURSOR statement. If INSENSITIVE was specified in DECLARE CURSOR, INSENSITIVE is the default in the FETCH statement. If SENSITIVE was specified in DECLARE CURSOR, SENSITIVE is the default in the FETCH statement.

**Note 2:** If SENSITIVE or INSENSITIVE is specified in the FETCH statement, *single-row-fetch* must also be specified.

**Note 3:** If BEFORE or AFTER is specified, do not specify SENSITIVE, INSENSITIVE, or the accompanying *single-row-fetch*.

**Note 4:** For a scrollable cursor, all FETCH statements with the *single-row-fetch* must specify the same number of columns.

**Note 5:** INTO and USING are optional, as shown, if a SCROLL CURSOR is being used. For non-SCROLL CURSOR operations, however, specifying either INTO or USING is required.

## Description

### Positioning

Positioning is either BEFORE or AFTER the first or last row of the result set, or it is based on the ABSOLUTE or RELATIVE position in the result set.

The position value must be an integer literal or host variable.

### ABSOLUTE

The cursor is positioned to the row specified from the beginning, or end if negative, of the result set. If the position is zero, the cursor is positioned before the first row of the set. If the position value is greater than the number of rows in the result set, a warning is issued and the position is changed to after the last row.

Examples with a set of 100 rows:

0 - cursor is positioned before first row of set, no data returned.

10 - cursor is positioned on the tenth row of the set, data returned (unless a hole with the static sensitive cursor mode).

-1 - cursor is positioned on the last row of the set.

101 - cursor is positioned after the last row of the set, no data returned, warning issued.

-101 - cursor is positioned before the first row of the set, no data returned, warning issued.

### RELATIVE

The cursor is positioned from the current position. A negative value positions backwards, and a positive value forwards. Zero returns the current row; however, if the current row has been deleted or updated such that it no longer qualifies, then the next row of the set is returned for the dynamic sensitive cursor mode.

0 - the same row is returned. If it has been deleted or no longer qualifies for a sensitive dynamic cursor, then the next row is returned.

+10 - cursor is moved forward 10 rows, and data returned (unless a hole with the static sensitive cursor mode). If position is after the last row of the set, a warning is returned and no data returned.

-1 - cursor is positioned backwards one row.

+101 - cursor is positioned after the last row of the set, no data returned, warning issued.

-101 - cursor is positioned before the first row of the set, no data returned, warning issued.

Cursors are declared insensitive, sensitive static and sensitive dynamic, but there is also the option of specifying insensitive or sensitive on the fetch.

<b>Declare</b>	<b>Fetch Insensitive</b>	<b>Fetch Sensitive</b>
<b>Insensitive</b>	Not needed since it is the default.	Error returned; no effect on cursor
<b>Sensitive Dynamic</b>	Error returned; no effect on cursor.	Row is returned from the database, reflecting changes made by this transaction, and other committed changes.
<b>Sensitive Static</b>	Returns row as is	<ul style="list-style-type: none"> <li>■ If the row has been deleted, no data is returned.</li> <li>■ If the row has been updated such that it no longer qualifies for the set, no data is returned.</li> <li>■ If the row has been updated, the updated values are returned.</li> <li>■ Rows inserted since the cursor was opened that would qualify are not visible.</li> </ul>
<b>Any</b>	<ul style="list-style-type: none"> <li>■ Positioned update or delete from the cursor are reflected.</li> <li>■ Previous fetch sensitive reflected.</li> </ul>	

**Starting and Resulting Cursor Position**

Kalpana Shyam of IBM Silicon Valley Lab provided the table below in her presentation "Scrollable Cursors: Fetching Opportunities for DB2 for OS/390" at the DB2 and Business Intelligence Technical Conference, October 16-20, 2000.

**Starting and resulting cursor position**

Fetch Orientation	Current Position...				Resulting Position...				
	Before First Row	On First Row	On last Row	After Last Row	On Delete Hole	On Update Hole	On Normal Row	Before First Row	After Last Row
NEXT	OK	OK	+100	+100	+222	+222	IF OK	IF +100 FROM FIRST ROW	IF +100 FROM LAST ROW
PRIOR	+100	+100	OK	OK	+222	+222	IF OK	IF +100 FROM LAST ROW	IF +100 FROM FIRST ROW
FIRST	OK	OK	OK	OK	+222	+222	IF OK	N/A	N/A
LAST	OK	OK	OK	OK	N/A	N/A	IF OK	N/A	N/A
BEFORE,	OK	OK	OK	OK	N/A	N/A	N/A	IF OK	N/A
AFTER	OK	OK	OK	OK	+222	+222	N/A	N/A	IF OK
CURRENT	+231	OK	OK	+231	+222	+222	IF OK	N/A	N/A
RELATIVE 0									
ABSOLUTE +N	OK	OK	OK	OK	+222	+222	IF OK	N/A	IF +100 AND N OUT OF RANGE
ABSOLUTE -N	OK	OK	OK	OK	+222	+222	IF OK	IF +100 AND N OUT OF RANGE	N/A
RELATIVE +N	OK	OK	+100	+100	+222	+222	IF OK	N/A	IF +100 AND N OUT OF RANGE
RELATIVE -N	+100	+100	OK	OK	+222	+222	IF OK	IF +100 AND N OUT OF RANGE	N/A

**Note:** If a fetch encounters an update or delete hole, a +222 SQLCODE is returned to the program.

**SQLCA**

**Explanation of SQLCA fields on scrollable cursors**

Field	Value	Meaning
SQLWARN1	N	Non-scrollable
SQLWARN1	S	Scrollable
SQLWARN4	I	Insensitive
SQLWARN4	S	Sensitive static
SQLWARN5	1	Read-only implicitly or explicitly
SQLWARN5	2	Select and delete allowed on result table

Field	Value	Meaning
SQLWARN5	3	Select, delete, and update allowed on result table

***cursor-name***

Specify the name of a cursor that is defined in a DECLARE CURSOR statement of your program. The DECLARE CURSOR statement must precede the FETCH statement in your source program. When the FETCH statement is executed, the cursor must be in the open state.

***INTO host-variable***

If INTO is used, each host-variable you specify must identify a variable that is described in your program in accordance with the rules for declaring host variables. The host variables must be separated by commas.

The first value in the result row is assigned to the first variable in the list, the second value to the second variable, and so on. If the number of variables is not the same as the number of values in the row, the SQLCA-WARNING(4) field of the SQLCA is set to W.

The data type of a variable must be compatible with the value assigned to it. If the value is numeric, the variable must have the capacity to represent the integral part of the value. If the value is null, an indicator variable must be specified.

Each assignment to a variable is made according to the rules described in Basic Operations (Assignment and Comparison) Assignments are made in sequence through the list. If an assignment error occurs, the value is not assigned to the variable, and no more values are assigned to variables. Any values that have already been assigned to variables remain assigned.

***USING DESCRIPTOR descriptor-name***

This clause allows the row of a result table of a cursor to be fetched into variables which are determined at execution-time. Descriptor-name identifies a SQL Descriptor Area (SQLDA) that contains a valid description of zero or more host variables. The length of the SQLDA, as indicated by SQLABC, must be sufficient to describe the number of variables indicated by SQLD. The first value of a row corresponds to the first variable described by the SQLDA, the second value to the second variable, and so on. For more information about the SQLDA, see [SQL Descriptor Area \(SQLDA\)](#) (see page 869).

**Note:** A cursor referenced in an UPDATE or DELETE statement must be positioned on a row. A cursor can only be positioned on a row as a result of a FETCH statement. If an error occurs during the execution of a FETCH statement that makes the position of a cursor unpredictable, the cursor is closed.

**Example**

See the DECLARE CURSOR's "Example 1" on [Example1](#) (see page 715).

## GRANT

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
GRANT	YES	YES	YES

**Note:** YES indicates a valid execution method for this statement. To learn about using SQL keywords in CA Dataquery, see the *CA Dataquery User Guide*. For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

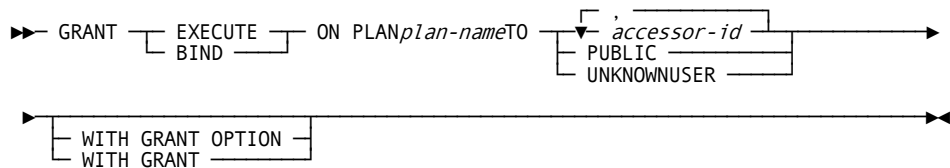
The GRANT statement grants privileges on tables and views in databases that are secured under the SQL Security Model. All of the tables and views specified in the GRANT statement must be in databases secured under the SQL Security Model. If tables and views are in databases secured under the CA Datacom/DB External Security Model, the GRANT statement is rejected with an SQL error code -273. See the *CA Datacom Security Reference Guide* for more information about security models.

## Plan Security

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 [REVOKE](#) (see page 759), Also see the information on the CHECKPLAN=, CHECKWHEN=, CHECKWHO=, and SAVEPLANSEC= options. For detailed information about plan security, see the *CA Datacom Security Reference Guide*.

**Note:** To grant a plan privilege you must possess that privilege WITH GRANT OPTION or be a Global Owner. To revoke a plan privilege you must have granted the privilege or be a Global Owner. See the *CA Datacom Security Reference Guide* for more information on Global Owners.

Following is the syntax diagram for the plan security version of the GRANT statement:



## Description of Plan Security Diagram

**EXECUTE**

Grants the PLAN EXECUTE privilege.

**BIND**

Grants the PLAN BIND privilege.

**ON PLAN *plan-name***

Specifies the name of the plan to which the PLAN EXECUTE or PLAN BIND privilege is to be granted.

**TO *accessor-id***

Specify the accessor ID of one or more users to whom you are granting the privileges. This is a *user's* ID, not a schema auth-id. Do not specify your own accessor ID, that is to say, you cannot grant privileges to yourself.

**TO PUBLIC**

Specify PUBLIC when you are granting the specified privileges to all users. A new user automatically has any privileges previously granted to the public.

**TO UNKNOWNUSER**

Specify UNKNOWNUSER when you are granting the specified privileges to users whose identities cannot be determined by CA Datacom/DB security.

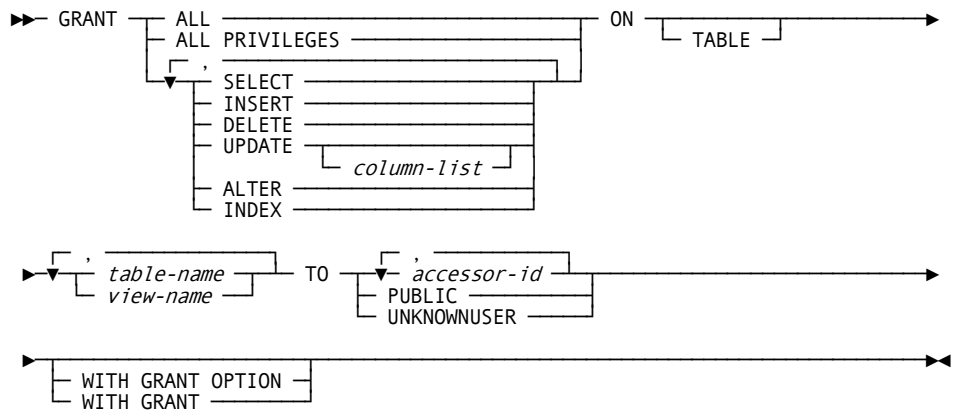
**WITH GRANT OPTION**

Specify this option if you want the user to whom you have granted the privilege to be able to grant it to another user. WITH GRANT OPTION cannot be used with PUBLIC.

**WITH GRANT**

Specify WITH GRANT if you want the user to whom you have granted the privilege to be able to grant it to another user. WITH GRANT cannot be used with PUBLIC or with UNKNOWNUSER.

Following is the non-plan security syntax diagram for the GRANT statement:



The following are CA Datacom/DB extensions.

- ALL
- TABLE
- ALTER
- INDEX
- UNKNOWNUSER
- WITH GRANT (without the keyword OPTION)

## Description of Non-Plan Security Diagram

**ALL or ALL PRIVILEGES**

Grants all privileges (excluding ALTER and INDEX) for which you have GRANT authority on all tables or views named in the ON clause. GRANT ALL is a CA Datacom/DB extension.

If you do not use ALL, you must use one or more of the following keywords. Each keyword grants the privilege described, but only as it applies to the tables or views named in the ON clause.

**SELECT**

Grants the privilege to use the SELECT statement.

**INSERT**

Grants the privilege to use the INSERT statement.



**DELETE**

Grants the privilege to use the DELETE statement.

**ALTER**

Grants the privilege to use the ALTER statement. GRANT ALTER is a CA Datacom/DB extension.

**INDEX**

Grants the privilege to execute the CREATE INDEX and DROP INDEX statements. GRANT INDEX is a CA Datacom/DB extension.

**UPDATE**

Grants the privilege to use the UPDATE statement.

**UPDATE (*column-list*)**

Grants the privilege to update only the named columns. Each column-name must belong to every table or view named in the ON clause. The column names must be separated by commas and the list must be enclosed with parentheses.

**ON or ON TABLE**

Introduces a list of table and/or view names. ON TABLE is a CA Datacom/DB extension.

***table-name or view-name***

Specify the name of one or more tables or views on which you are granting the privileges. The list can be a list of table names or view names, or a combination of the two. The names must be separated by commas.

If you name specific privileges, then for each table or view you name, you must have all those privileges with GRANT authority. If you use GRANT ALL, then for each table or view you name, you must have at least one privilege with GRANT authority.

**TO *accessor-id***

Specify the accessor ID of one or more users to whom you are granting the privileges. This is a *user's* ID, not a schema auth-id. If listing more than one accessor ID, separate them with commas. Do not specify your own accessor ID, that is to say, you cannot grant privileges to yourself.

**TO PUBLIC**

Specify PUBLIC when you are granting the specified privileges to all users. A new user automatically has any privileges previously granted to the public.

**TO UNKNOWNUSER**

Specify UNKNOWNUSER when you are granting the specified privileges to users whose identities cannot be determined by CA Datacom/DB security. UNKNOWNUSER is a CA Datacom/DB extension.

**WITH GRANT OPTION**

Specify this option if you want the user to whom you have granted the privilege to be able to grant it to another user. The WITH GRANT OPTION cannot be used with PUBLIC.

**WITH GRANT**

Specify WITH GRANT if you want the user to whom you have granted the privilege to be able to grant it to another user. WITH GRANT cannot be used with PUBLIC or with UNKNOWNUSER. WITH GRANT is a CA Datacom/DB extension.

Example 1

Grant SELECT privileges on table TEMPL to user PULASKI.

```
GRANT SELECT
  ON TABLE CA.TEMPL
  TO PULASKI
```

Example 2

Grant UPDATE privileges on columns EMPNO and WORKDEPT in table TEMPL to all users.

```
GRANT UPDATE (EMPNO, WORKDEPT)
  ON TABLE CA.TEMPL
  TO PUBLIC
```

Example 3

Grant all privileges on table TEMPL to users KWAN and THOMPSON, including the WITH GRANT OPTION.

```
GRANT ALL PRIVILEGES
  ON TABLE CA.TEMPL
  TO KWAN, THOMPSON
  WITH GRANT OPTION
```

## IF-THEN Statement

For details about this statement, see [IF-THEN Statement](#) (see page 652).

**Note:** This statement is executed from within the Compound Statement (see [Compound Statement](#) (see page 642)).

# INSERT

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
INSERT	YES	YES	YES

**Note:** YES indicates a valid execution method for this statement. To learn about using SQL keywords in CA Dataquery, see the *CA Dataquery User Guide*. For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

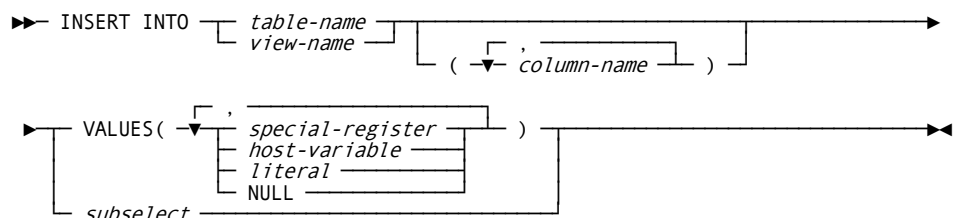
The INSERT statement inserts rows into a table or view. Inserting a row into a view inserts the row into the table upon which the view is based. You must specify ISOLEVEL=C (isolation level C) in the Preprocessor options when using the INSERT statement.

**Note:** When the NOMAINT option of the CA Datacom/DB Utility (DBUTLTY) ACCESS function is in force, an INSERT statement receives a CA Datacom/DB return code 94(87), where 87 is a decimal internal return code (hex 57) that tells you no maintenance statements are allowed while NOMAINT is in force.

Also note that constraints that existed in versions before r10 can act differently in Release 10.0 and above because, in Release 10.0 and above, columns containing a NULL rather than a value do *not* cause a CHECK constraint to be violated. Constraints are considered, in Release 10.0 and above, to be satisfied unless the predicates evaluate explicitly to FALSE. That is, CHECK constraints whose predicates evaluate to UNKNOWN rather than TRUE or FALSE are considered, with Release 10.0 and above, to have been satisfied. Therefore, INSERT and UPDATE statements, from versions before r10, that resulted in constrained columns being nulled are, when used in Release 10.0 and above, successful for the first time.

Following is the syntax diagram for the INSERT statement. See [Special Registers](#) (see page 533) for the special-register diagram. See [Subselect](#) for the subselect's syntax diagram.

**Note:** The special-register is a CA Datacom/DB extension.



## Description

### ***table-name or view-name***

Specify the name of the table or view into which you want to insert the row(s). The table or view must be described in the CA Datacom Datadictionary, and must not be a CA Datacom Datadictionary table or any of the following types of views:

- A read-only view.
- A view of a CA Datacom Datadictionary table.
- A view with a column that is derived from a literal or an arithmetic expression.
- A view with two columns derived from the same column of the underlying table.

### ***column-name***

Specify the name of one or more columns for which you provide insert values. You can name the columns in any order. Each column must belong to the table or view you specified, and you cannot name the same column more than once. The column names must be separated by commas and the list must be enclosed with parentheses.

### **VALUES**

Introduces one row of values to be inserted. The values of the row are the values of the keywords, literals, or host variables specified in the clause. The values must be separated by commas and the list must be enclosed with parentheses.

When the statement is executed, the number of values in the VALUES clause must equal the number of names in the column list. The first value is inserted in the first column in the list, the second value in the second column, and so on.

### ***special-register***

See Special Registers for the special-register diagram. This is a CA Datacom/DB extension.

### ***host-variable***

Specify a host-variable which must be a variable that is described in your program in accordance with the rules for declaring host variables.

### ***literal***

Specify a literal consistent with the data type of the column.

**NULL**

Use the NULL keyword to specify that the value(s) being inserted are null values.

***subselect***

If you specify a subselect, the rows of the subselect's result table are inserted into the table or view you specified in the INSERT statement. The result can be one row, more than one row or no rows. If no rows are inserted, SQLCODE is set to +100. See Subselect for the subselect's syntax diagram.

The table or view named after INSERT INTO must not be named in a FROM clause of the subselect or the FROM clause of any subquery in the subselect.

The number of columns in the result table must equal the number of names in the column list. The value of the first column of the result is inserted in the first column in the list, the second value in the second column, and so on.

## Rules for Inserting

Insert values must satisfy the following rules:

**Default values:**

The value inserted in any column not in the column list is the default value of the column. Columns without a default value must be included in the column list.

If you insert into a view, the default value is inserted into any column of the base table that is not included in the view.

All columns of the base table that are not in the view must have default values.

**Length:**

If the insert value of a column is a number, the column must be a numeric column with the capacity to represent the integral part of the number.

If the insert value of a column is a string, the column must be a string column with a length attribute at least as great as the length of the string.

If insert values do not adhere to the previously given rules, or if any other error occurs during the execution of the INSERT statement, no rows are inserted.

## Processing

Rows can be inserted that do not conform to the definition of a view. These rows cannot appear in the view, but are inserted into the base table of the view.

Unless appropriate locks already exist, one or more exclusive locks are acquired at the execution of a successful INSERT statement. Until the locks are released, an inserted row can be accessed only by the unit of recovery in which the insert was performed.

### Example 1

Insert a row into table EMP.

```
EXEC SQL
    INSERT INTO EMP
        VALUES('315', 'JOHN', 'T', 'SMITH', 'A1', '1234',
            87 11, 32, 19, 'M', 55 422, 16325)
END-EXEC
```

### Example 2

Load the temporary table TEMPEMP with data from table EMP.

```
EXEC SQL
    INSERT INTO SMITH.TEMPEMP
    SELECT _
    FROM EMP
END-EXEC
```

### Example 3

Load the temporary table TEMPEMP with data from Department E1 from EMP.

```
EXEC SQL
    INSERT INTO SMITH.TEMPEMP
    SELECT FROM EMP
    WHERE WORKDEPT = 'E1'
END-EXEC
```

## ITERATE Statement

For details about this statement, see [ITERATE Statement](#) (see page 653).

**Note:** This statement is executed from within the Compound Statement (see [Compound Statement](#) (see page 642)).

## LEAVE Statement

For details about this statement, see [LEAVE Statement](#) (see page 654).

**Note:** This statement is executed from within the Compound Statement (see [Compound Statement](#) (see page 642)).

## LOCK TABLE

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
LOCK TABLE	YES	YES	

**Note:** YES indicates a valid execution method for this statement. For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

CA Datacom/DB provides the LOCK TABLE statement as an extension so you can acquire a shared or exclusive lock on the named table. The table lock is acquired when the LOCK TABLE statement is executed. The table lock is released at the next commit point.

With regard to table partitioning, a LOCK TABLE statement can receive a CA Datacom/DB return code of 01(034) if it attempts to use a Any Parent table. See the *CA Datacom/DB Message Reference Guide* for details about return code 01(034). Information about table partitioning can be found in the *CA Datacom/DB Database and System Administration Guide*.

Following is the syntax diagram for the LOCK TABLE statement:

```

▶▶ LOCK TABLE table-name IN ┌ SHARE ───────────┐ MODE ───────────────────────────────────▶▶
                               │ EXCLUSIVE │

```

## Description

### ***table-name***

Specify the name of the table to be locked. The table must be a base table described in the CA Datacom Datadictionary, but not a CA Datacom Datadictionary table.

### **IN SHARE MODE**

Acquires a shared lock for the unit of recovery in which the statement is executed. The lock prevents other concurrent units of recovery from inserting, updating or deleting rows in the identified table.

Using the SHARE mode of the LOCK TABLE statement ensures that the data retrieved by your application is valid, that is to say, has been committed by a previous transaction.

**IN EXCLUSIVE MODE**

Acquires an exclusive lock for the unit of recovery in which the statement is executed. The lock prevents other concurrent units of recovery from acquiring any shared or exclusive row or table lock.

Using either mode of the LOCK TABLE statement ensures that your application can do a repeatable read. This isolation level provides maximum protection from other executing application programs. When a program executes with repeatable read protection, rows referenced by the program cannot be changed by other programs until the program reaches a commit point.

Neither lock prevents rows from being read with user isolation (ISOLEVEL=U in Preprocessor options) since no locks are acquired by user isolation.

Example

The following example:

- Obtains a lock on the table containing table NAMETBL.
- Does not allow another program to update the table.

```
EXEC SQL
    LOCK TABLE NAMETBL IN EXCLUSIVE MODE
END-EXEC
```

LOOP Statement

For details about this statement, see [LOOP Statement](#) (see page 655).

**Note:** This statement is executed from within the Compound Statement (see [Compound Statement](#) (see page 642)).

OPEN

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
OPEN		YES	





**USING**

The USING clause introduces a list of host variables whose values are substituted for the parameter markers of the prepared select-statement. If the DECLARE CURSOR statement names a prepared statement that includes parameter markers, you must use USING. If the prepared statement does not include parameter markers, USING is ignored.

For more information on the USING clause, see the discussion about it in the section on the EXECUTE statement.

The USING clause is intended for a prepared select-statement that contains parameter markers. However, it can also be used when the select-statement of the cursor is part of the DECLARE CURSOR statement. In this case the OPEN statement is executed as if each host variable in the select-statement were a parameter marker. Thus the effect is to override the host variables in the select-statement of the cursor with the host variables specified in the USING clause.

***host-variable***

A reference to each of its variables replaces reference to a structure when the statement is executed, where the number of variables must equal the number of parameter markers in the prepared statement, and the *n*th variable corresponds to the prepared statement's *n*th parameter marker.

**Note:** The OPEN is rejected with SQL error code -305 if the number of variables does not equal the number of parameter markers of host variables in the SELECT statement.

**Parameter Marker Replacement:** Each parameter marker in the query is effectively replaced by its corresponding host variable before the OPEN statement is executed. The replacement is an assignment operation in which the source is the value of the host variable and the target is a variable within CA Datacom/DB. The attributes of the target variable operand are the same as the left-most column reference operand or, if no column reference operand exists, the first non-parameter marker operand. If there are no operands, or if they are all parameter markers, or if the operator is concatenated, a parameter marker cannot be used, and the statement is rejected with SQL error code -302.

**DESCRIPTOR *descript-name***

The value to which SQLD is set is required to be:

- Equal to or greater than zero, and
- Less than or equal to the value of SQLN, and
- The same as the number of parameter markers in the prepared statement.

**Note:** The *n*th variable described by the SQLDA corresponds to the prepared statement's *n*th parameter marker.

## Processing

All cursors in a program are in the closed state when:

1. The program is initiated.
2. The program initiates a new unit of recovery by executing a COMMIT or ROLLBACK operation.

**Note:** Unless the WITH HOLD option has been used in the DECLARE CURSOR statement, a new unit of work is started for the application process.

A cursor can also be in the closed state because:

1. A CLOSE statement was executed.
2. An error was detected that made the position of the cursor unpredictable.

The only way to change the state of a cursor from closed to open is to execute an OPEN statement.

To retrieve rows from the result table of a cursor, you must execute a FETCH statement when the cursor is open.

## Effect of Temporary Tables

In some cases, CA Datacom/DB derives the result table of a cursor by first creating a temporary table when the OPEN statement is executed. When a temporary table is used, the results of a program can differ in two ways:

1. An error can occur during OPEN that would otherwise not occur until some later FETCH statement.
2. If you selected cursor stability as your isolation level, INSERT, UPDATE, and DELETE statements executed by other transactions while the cursor is open do not affect the result table.

## Example

See the DECLARE CURSOR's "Example 1" on [Example1](#) (see page 715).

# PREPARE

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
PREPARE		YES	

**Note:** YES indicates a valid execution method for this statement.

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

Prepared statements are deleted when the unit of recovery in which they were prepared ends, except that a select-statement whose cursor is declared with the WITH HOLD option persists (if the cursor is open) over a COMMIT WORK.

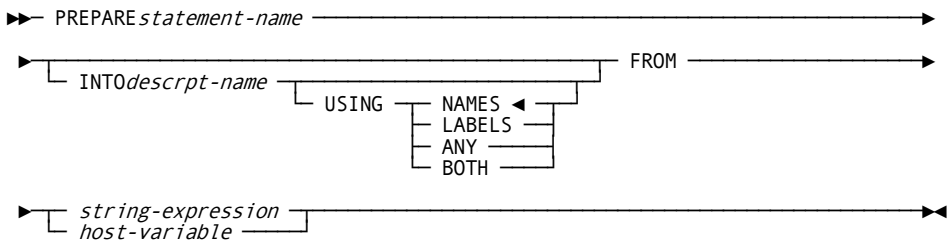
No authorization is required to preprocess a PREPARE statement, that is, there is no security check of the PREPARE statement or its contents at preprocessor time. At execution time, however, the statement's contents *are* checked against the security conditions that are valid for that program.

Prepared statements can be referenced by these statements:

- DESCRIBE
- DECLARE CURSOR (must be a select-statement when cursor is opened)
- EXECUTE (must not be a select-statement)

A prepared statement can be executed multiple times.

Following is the syntax diagram for the PREPARE statement:



---

## Description

***statement-name***

Specifies the name under which the executable form of the statement (the prepared statement) is saved if there are no errors. If the name identifies an existing prepared statement, that statement is deleted. The prepared statement must not be the select-statement of an open cursor. The scope of *statement-name* is the same as the scope of *cursor-name*.

**INTO**

If the prepare is successful, information about the prepared statement is placed in the SQLDA specified by *descript-name*, as if you had executed a separate DESCRIBE statement.

***descript-name***

Identifies the SQLDA (descriptor name). SQLN should be set to the number of SQLVAR occurrences.

**USING**

Specifies if the SQLNAME field is to contain a column name or label.

**NAMES**

Assigns the name of the column. This is the default.

**LABELS**

Assigns the label of the column. A label is the column's CA Datacom Datadictionary field attribute HEADING-1.

**ANY**

Assigns the column label, or (if one does not exist) the column name.

**BOTH**

Assigns the column label to the first *n* occurrences, and the column label to the second *n* occurrences.

**FROM**

Specifies the statement string, which is the value of the string-expression or host-variable.

***string-expression***

A PL/I expression that yields a character string.

***host-variable***

A host-variable must be used for languages other than PL/I and identify a varying-length string variable.

## Rules for Statement Strings

A statement string is the value of the string-expression or host-variable. A host-variable containing the statement string must be a varying-length string variable. The statement string must not begin with EXEC SQL nor end with a statement terminator, and it must not include references to host variables.

<b>The statement string can be any of the following statements:</b>	<b>The statement string must not be any of the following statements:</b>
ALTER TABLE	CALL
COMMENT ON	CLOSE
COMMIT WORK	DECLARE CURSOR
CREATE	DECLARE STATEMENT
DELETE	DESCRIBE
DROP	EXECUTE
GRANT	EXECUTE IMMEDIATE
INSERT	EXECUTE PROCEDURE
LOCK TABLE	FETCH
REVOKE	OPEN
ROLLBACK WORK select-statement	PREPARE
UPDATE	SELECT (other than the select-statement) SET
	CURRENT SQLID
	WHENEVER

## Rules for Parameter Markers

A parameter marker is a question mark (?) that is used in place of a host variable in dynamic statements. Parameter markers must *not* be used:

- In an ESCAPE clause.
- In a select list.
- As an operand of the concatenation operator.
- As both operands of an arithmetic or comparison operator.
- As an operand in a date/time arithmetic expression.
- As the first operand of a LIKE predicate.
- As the first operand of a NULL predicate.

At least one of the operands of a BETWEEN or IN predicate must *not* be a parameter marker.

An argument of a scalar function *cannot* be specified solely as a parameter marker.

A parameter marker *can* be the operand of a unary minus operator.

## Example

```

01 S1.
   49 S1LEN          PIC S9999 COMP VALUE +80.
   49 S1HV           PIC X(8) .
01 HV1              PIC X VALUE 'X' .
01 HV2              PIC X VALUE 'Y' .

MOVE 'INSERT INTO T1 VALUES (?, ?)' TO S1HV.
EXEC SQL
    PREPARE STMT1 FROM S1
END-EXEC
(Check for successful prepare and set host variable values.)
EXEC SQL
    EXECUTE STMT1 USING :HV1, :HV2
END-EXEC

```

## REPEAT-UNTIL Statement

For details about this statement, see [REPEAT-UNTIL Statement](#) (see page 658).

**Note:** This statement is executed from within the Compound Statement (see [Compound Statement](#) (see page 642)).

## REVOKE

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
REVOKE	YES	YES	YES

**Note:** YES indicates a valid execution method for this statement. To learn about using SQL keywords in CA Dataquery, see the *CA Dataquery User Guide*. For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

**Important!** If CA Datacom/DB security is not installed at your site, the REVOKE statement is rejected with an SQL error code -559.

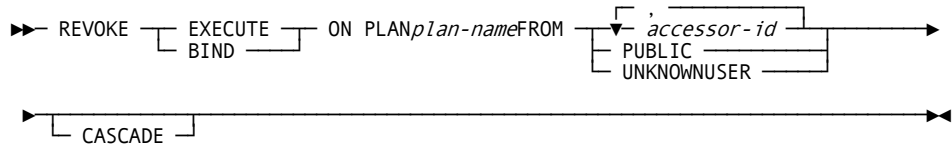
CA Datacom/DB provides the REVOKE statement as an extension so you can revoke privileges on tables and views for which privileges have been granted through the GRANT statement. All tables and views in the statement must belong to databases secured under the SQL Security Model. If tables and views are in databases secured under the CA Datacom/DB External Security Model, the GRANT and REVOKE statements are rejected with an SQL error code -273. See the *CA Datacom Security Reference Guide* for more information about security models.

## Plan Security

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. Also see the information on the REVOKE statement and the CHECKPLAN=, CHECKWHEN=, CHECKWHO=, and SAVEPLANSEC= options. For detailed information about plan security, see the *CA Datacom Security Reference Guide*.

**Note:** To grant a plan privilege you must possess that privilege WITH GRANT OPTION or be a Global Owner. To revoke a plan privilege you must have granted the privilege or be a Global Owner. See the *CA Datacom Security Reference Guide* for more information on Global Owners.

Following is the syntax diagram for the plan security version of the REVOKE statement:



## Description of Plan Security Diagram

### EXECUTE

Revokes the PLAN EXECUTE privilege.

### BIND

Revokes the PLAN BIND privilege.

### ON PLAN plan-name

Specifies the name of the plan from which the PLAN EXECUTE or PLAN BIND privilege is to be revoked.

### FROM accessor-id

Specify the accessor ID of a user from whom you are revoking the privileges that were granted with a GRANT statement. This is a user's ID, not a schema auth-id.



**FROM PUBLIC**

Specify PUBLIC when you are granting or revoking the specified privileges to or from all users. A new user automatically has any privileges previously granted to the public.

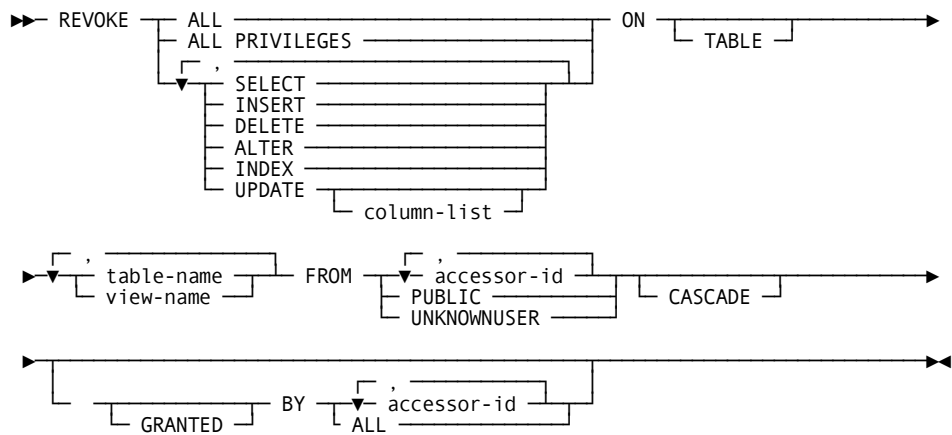
**FROM UNKNOWNUSER**

Specify UNKNOWNUSER when you are revoking the specified privileges from users whose identities cannot be determined by the CA Datacom/DB security.

**CASCADE**

If CASCADE is specified, any other dependent privileges that have been granted to others (through the GRANT statement) are also revoked. If a REVOKE is issued without CASCADE and the grantee granted privileges to other users, the REVOKE is not permitted. The CASCADE option of REVOKE does not block the cascading effect of a revoke but operates instead as a fail-safe device. Specifying CASCADE simply acknowledges your understanding that there are cascading effects.

Following is the non-plan security syntax diagram for the REVOKE statement:



## Description of Non-Plan Security Diagram

### **ALL or ALL PRIVILEGES**

Revokes all privileges (excluding ALTER and INDEX) which the executor of the REVOKE statement has previously granted to the specified users.

If you do not use ALL, you must use one or more of the following keywords. Each keyword revokes the privilege described, but only as it applies to the tables or views named in the ON clause.

### **SELECT**

Revokes the privilege to use the SELECT statement.

### **INSERT**

Revokes the privilege to use the INSERT statement.

### **DELETE**

Revokes the privilege to use the DELETE statement.

### **ALTER**

Revokes the privilege to use the ALTER statement.

### **INDEX**

Revokes the privilege to execute the CREATE INDEX and DROP INDEX statements.

### **UPDATE**

Revokes the privilege to use the UPDATE statement.

### **UPDATE (*column-name*)**

Revokes the privilege to update only the named columns. Each column-name must belong to every table or view named in the ON clause. The column names must be separated by commas and the list must be enclosed with parentheses.

### **ON or ON TABLE**

Introduces a list of table and/or view names.

### ***table-name or view-name***

Specify the name of one or more tables or views on which you are revoking the privileges. The list can be a list of table names or view names, or a combination of the two. The names must be separated by commas.

For each table or view you identify, you (or the indicated grantors) must have granted (using the GRANT statement) at least one of the specified privileges on that table or view to all identified users (including PUBLIC, if specified).

### **FROM *accessor-id***

Specify the accessor ID of one or more users from whom you are revoking the privileges that were granted with a GRANT statement. This is a *user's* ID, not a schema auth-id. If listing more than one accessor ID, separate them with commas.

**FROM PUBLIC**

Specify PUBLIC when you are revoking the specified privileges from all users.

**FROM UNKNOWNUSER**

Specify UNKNOWNUSER when you are revoking the specified privileges from users whose identities cannot be determined by the CA Datacom/DB security.

**CASCADE**

If CASCADE is specified, any other dependent privileges that have been granted to others (through the GRANT statement) are also revoked. If a REVOKE is issued without CASCADE and the grantee granted privileges to other users, the REVOKE is not permitted. The CASCADE option of REVOKE does not block the cascading effect of a revoke but operates instead as a fail-safe device. Specifying CASCADE simply acknowledges your understanding that there are cascading effects.

**GRANTED**

Allows you to specify that you are revoking privileges that were GRANTED BY another user(s). The accessor ID(s) specified in GRANTED BY must therefore have previously granted the specified privileges to the grantee. This form of the REVOKE statement may only be executed by a global database owner. For information about global database owners, see the *CA Datacom Security Reference Guide*.

**BY accessor-id**

The BY indicates that the person revoking the privileges is doing so on behalf of another user. Specify the accessor ID of the person who granted the privileges you are revoking. The accessor IDs must be separated by commas.

**BY ALL**

Specify ALL when revoking privileges granted by all other users to the user identified in the FROM clause of the REVOKE statement.

## Example 1

Revoke SELECT privileges on table TEMPL from user PULASKI.

```
REVOKE SELECT
  ON TABLE CA.TEMPL
  FROM PULASKI
```

### Example 2

Revoke UPDATE privileges on table TEMPL previously granted to all users.

**Note:** Grants to specific users are not affected.

```
REVOKE UPDATE
  ON TABLE CA.TEMPL
  FROM PUBLIC
```

### Example 3

Revoke all privileges on table TEMPL from users KWAN and THOMPSON.

```
REVOKE ALL PRIVILEGES
  ON TABLE CA.TEMPL
  FROM KWAN, THOMPSON
```

## ROLLBACK WORK

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
ROLLBACK WORK	YES	YES	

**Note:** YES indicates a valid execution method for this statement. For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

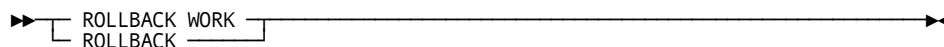
The ROLLBACK WORK statement terminates a unit of recovery and backs out the CA Datacom/DB database changes made by that unit of recovery.

If a cursor is defined WITH HOLD, it stays open when a COMMIT WORK is executed. Any record-at-a-time command that commits the logical unit of work (for example LOGCP, LOGCR) works the same way. See the description of the WITH HOLD clause in DECLARE CURSOR.

Also, see the *CA Datacom/DB Database and System Administration Guide* for information on DIAGOPTION line 0,128,ON. This line can change the resulting cursor state (open or closed) following a ROLLBACK WORK statement when SQLMODE=DATACOM is specified. This option is not recommended for general use.

Following is the syntax diagram for the ROLLBACK WORK statement:

**Note:** ROLLBACK (without the keyword WORK) is a CA Datacom/DB extension.



## Description

### ROLLBACK WORK

The ROLLBACK WORK statement is the rollback operation. The unit of recovery in which the statement is executed is terminated and a new unit of recovery is initiated. All changes made by CREATE, COMMENT ON, DROP, INSERT, UPDATE, and DELETE statements executed during the unit of recovery are backed out.

All locks implicitly acquired by the unit of recovery subsequent to its initiation are released.

### ROLLBACK

This CA Datacom/DB extension has the same effect as ROLLBACK WORK.

A unit of work is made up of 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.

A unit of recovery is a sequence of operations within a unit of work. A unit of recovery is initiated by:

1. The initiation of a unit of work.
2. The termination of a previous unit of recovery.

A unit of recovery is terminated by:

1. A commit operation.
2. A rollback operation.
3. The termination of a unit of work.

A commit or rollback operation affects only the results of SQL statements executed within a single unit of recovery.

Uncommitted database changes made by a unit of recovery may or may not be perceived by other units of work depending on the isolation level that is selected.

Uncommitted database changes made by a unit of recovery can be backed out by CA Datacom/DB.

Committed database changes can be perceived by other units of recovery and cannot be backed out by CA Datacom/DB.

Database changes are backed out when a unit of recovery terminates abnormally.

### Example

The following example deletes the alterations made since the last commit point or rollback:

```
EXEC SQL
      ROLLBACK WORK
END-EXEC
```

## SELECT

<b>This SQL statement can be executed in the following ways:</b>	<b>Through the CA Datacom Datadictionary Interactive SQL Service Facility (<i>interactive</i>)</b>	<b>In an application program prepared using a CA Datacom/DB SQL Preprocessor (<i>embedded</i>)</b>	<b>By using CA Dataquery (<i>SQL &amp; Batch Modes</i>)</b>
select-into statement		YES	
select-statement	YES	(use DECLARE CURSOR)	YES
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)

**Note:** YES indicates a valid execution method for this statement.

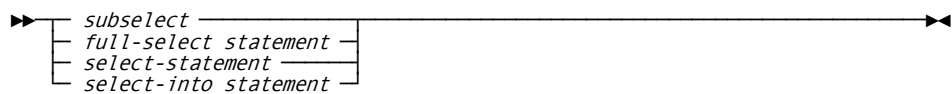
Information about the DECLARE CURSOR statement begins on DECLARE CURSOR.

**Note:** The subselect and full-select forms of the SELECT statement are not executable directly, because they are components of other statements. To learn about using SQL keywords in CA Dataquery, see the *CA Dataquery User Guide*. For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

The SELECT statement specifies a result table.

All of the tables and views specified in the SELECT statement must be in databases of the same security type, that is to say, either the CA Datacom/DB External Security Model or the SQL Security Model. See the *CA Datacom Security Reference Guide* for more information about security models.

Following is the syntax diagram for a SELECT statement:



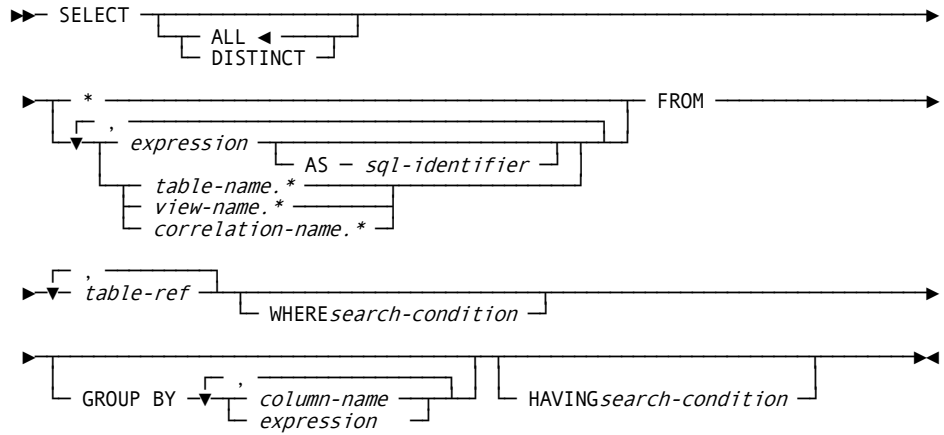
**Note:** The select-into statement is not used by CA Dataquery.

## Subselect

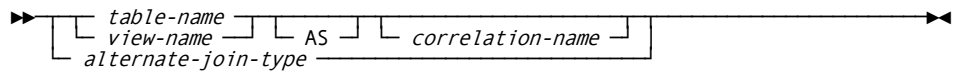
The subselect is a component of:

1. The full-select statement.
2. The CREATE VIEW statement.
3. The INSERT statement.
4. Certain predicates, which, in turn, are components of a subselect. A subselect that is a component of another subselect is called a subquery.

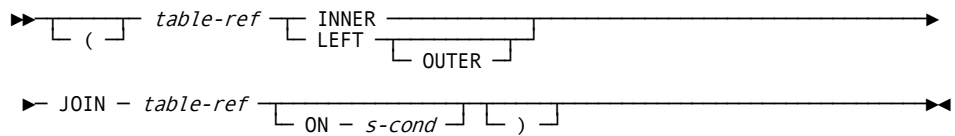
Following is the syntax diagram for the subselect statement:



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.



---

## Description

A subselect specifies a result table derived from the tables or views identified in the FROM clause. The derivation can be described as a sequence of operations in which the result of each operation is input for the next.

The sequence of the hypothetical operations is:

1. FROM clause
2. WHERE clause
3. GROUP BY clause
4. HAVING clause
5. SELECT clause

Descriptions of these clauses appear in the order as previously listed.

## FROM Clause

### **FROM table-ref**

Where table-ref can be a table-name, view-name, or alternate-join-type. If table-ref is a table-name or view-name, it names:

- A single table or view.
- Several tables and/or views to produce an intermediate result table.

If table-ref is an alternate-join-type, see the information about joins in [Left Outer Joins](#) (see page 110).

You can reference up to 20 tables in a FROM clause of a query when you are performing a join. For example, if a view is based on five tables, you can name that view in the FROM clause and up to 15 other tables. The names must be separated by commas.

The intermediate result table contains all possible combinations of the rows of the named tables or views. Each row of the result is a row from the first table or view concatenated with a row from the second table or view, concatenated in turn with a row from the third, and so on. The number of rows in the result is the product of the number of rows in all the named tables or views.

The list of names in the FROM clause must conform to these rules:

1. Each table-name and view-name must name a table or view described in the CA Datacom Datadictionary.
2. If the FROM clause is in a subquery of a basic predicate, no view named can use either GROUP BY or HAVING.

3. In other cases, if a named view uses GROUP BY, HAVING, or a column function, no other table or view can be named. The subselect statement is processed as if it contained the GROUP BY, HAVING, or column function used in the definition of the view.

***correlation-name***

The 1- to 18-byte *correlation-name* applies to the table or view named by the immediately preceding table-name or view-name. The correlation name can be used elsewhere in the statement to designate that table or view. Use a blank to separate the correlation name from the table-name or view-name.

## WHERE Clause

**WHERE *search-condition***

Produces an intermediate result table by applying the search-condition to each row of table R, which is the result of the FROM clause. The result table contains the rows of R for which the search condition is true.

**Note:** If using alternate-join-type, see the additional information about WHERE clauses in WHERE Clause. Also, in addition to the information about search conditions that follows, see the additional alternate-join-type search condition.

The search-condition describes a search condition that conforms to the following rules:

1. The condition is formed (see Search Conditions).
2. Each column-name in the search-condition either unambiguously identifies a column of R, or is a correlated reference. A correlated reference is allowed only in a subquery.
3. A column-name in the search-condition does not identify a column that is derived from a function or a grouping column. A column of a view can be derived from a function or a grouping column.
4. The search condition does not include a function unless the argument of the function is a correlated reference. This is only possible in a subquery of a HAVING clause.

Any subquery in the search-condition is effectively executed for each row of R and the results used in the application of the search-condition to the given row of R. A subquery is actually executed for each row of R only if it includes a correlated reference to a column of R.

---

## GROUP BY Clause

### **GROUP BY**

Produces an intermediate result table by grouping the rows of R, where R is the result of the previous clause.

#### ***column-name***

The column-name must unambiguously name a column of R. Each column named is called a group column. The names must be separated by commas.

The result of GROUP BY is a set of groups of rows. In each group of more than one row:

1. All values of each grouping column are equal.
2. All rows with the same set of values of the grouping columns are in the same group.

For the purpose of grouping, all null values within a grouping column are considered equal.

Because every row of a group contains the same value of any grouping column, the name of a grouping column can be used in a search condition in a HAVING clause or an expression in a SELECT clause. In each case, the reference specifies only one value for each group.

GROUP BY must not be used in:

- A subquery of a basic predicate.
- A subselect whose FROM clause names a view that used GROUP BY or HAVING.

#### ***expression***

GROUP BY elements can be expressions with the following restrictions:

- No column (aggregate) functions (SUM, AVG, and so on)
- Expression must reference a base table column in the FROM clause (that is, cannot have '5' or '10-2', but 'col1-5' is OK)
- No "case"
- No LOB column
- No special registers (CURRENT DATE, CURRENT TIME, and so on)

## HAVING Clause

### **HAVING *search-condition***

Produces an intermediate result table by applying the search-condition to each group of R, where R is the result of the previous clause. If that clause is not GROUP BY, all rows of R are considered as one group. The result table contains those groups of R for which the search condition is true.

The search-condition describes a search condition that conforms to the following rules:

1. The condition is formed (see Search Conditions).
2. Each column-name in the search-condition must:
  - a. Unambiguously identify a grouping column of R, or
  - b. Be a correlated reference, or
  - c. Be specified within a function.

A group of R to which the search condition is applied supplies the argument for each function in the search condition, except for any function whose argument is a correlated reference.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a group of R, and the results used in applying the search condition. In actuality, the subquery is executed for each group only if it contains a correlated reference.

A correlated reference to a group of R must either identify a grouping column or be contained within a function.

The HAVING clause must not be used in:

- A subquery of a basic predicate.
- A subselect whose FROM clause names a view that used GROUP BY or HAVING.

---

## SELECT Clause

### SELECT

Produces a final result table by selecting only the columns indicated by the select list from R, where R is the result of the previous clause.

**Note:** We do not impose an arbitrary limit on the number of columns you can select in a query. We are limited only by environmental factors. Some of these factors are the size of your work area as specified by the *size* parameter of the TASKS Multi-User startup option (see the *CA Datacom/DB Database and System Administration Guide*), your column sizes, and limits placed by other products such as the CA Datacom Server.

### ALL

Retains all rows of the final result table and does not eliminate redundant duplicates. ALL is the default.

### DISTINCT

Eliminates all but one of each set of duplicate rows of the final result table.

DISTINCT must not be used more than once in a subselect. This restriction includes:

1. Functions in the SELECT list of the subselect.
2. Functions in a HAVING clause of a subselect.
3. Functions which are specified in a subquery of the HAVING clause and contain a correlated reference to groups of the subselect.

### \*

The asterisk (\*) represents a list of names that identify the columns of R. The first name in the list identifies the first column of R, the second name identifies the second column of R, and so on. The list of names is established when the program containing the SELECT clause is prepared.

**Important!** SELECT \* and SELECT table.\* (see the following) are useful when selecting from tables or views in an interactive environment, especially when the names of the columns are not known. However, embedding SELECT \* or SELECT table.\* in an application program may cause unexpected results if the definition of the table is ever altered. For example, when a column is dropped from a table definition, all statements which reference the table are automatically rebound as they are executed. The rebound forms of the statements reflect the new set of columns. The application program, however, still expects the original set of columns, and the result table returned to the program no longer matches the FETCH statement's host variables. Views which are referenced by application programs should not include SELECT \* or SELECT table.\* for the same reason.

***name.\****

The asterisk (\*) represents a list of names that identify the columns of R. The **name** can be a table-name, view-name, or correlation-name, and must designate a table-name or view-name in the FROM clause. The first name in the list identifies the first column of R, the second name identifies the second column, and so on. The list is established at preparation time and does not represent any columns that have been added later. The names must be separated by commas.

***expression***

Commonly, the expressions used in a SELECT statement include column names. Each column name used in the select list must unambiguously identify a column of R. Multiple expressions must be separated by commas.

The number of columns in the result of SELECT is the same as the number of expressions in the operational form of the select list, that is to say, the list established at preparation time.

The result of a subquery must be a single column unless the subquery is used in the EXISTS predicate.

## Applying the Select List

Some of the results of applying the select list to R depend on whether GROUP BY or HAVING is or is not used.

**If neither GROUP BY nor HAVING is used:**

The select list must either include no functions, or be entirely a list of functions.

If the select includes no functions, then the select list is applied to each row of R, and the result contains as many rows as there are rows in R.

If the select list is a list of functions, then R is the source of the arguments of the functions, and the result of applying the select list is one row.

**If GROUP BY or HAVING is used:**

Each column-name in the select list must either identify a grouping column or be specified within a function.

The select list is applied to each group of R, and the result contains as many rows as there are groups in R. When the select list is applied to a group of R, that group is the source of the arguments of the functions in the select list.

In either case the nth column of the result contains the values specified by applying the nth expression in the operational form of the select list.

A result column derived from a column name acquires the unqualified name of that column. All other result columns have no names.

Each column of the result of SELECT acquires a data type from the expression from which it is derived.

### Example 1

Show all rows of the employee table TEMPL.

```
SELECT * FROM CA.TEMPL
```

### Example 2

Show the job code, maximum salary and minimum salary for each group of rows of the table TEMPL with the same job code, but only for groups with more than one row and with a maximum salary less than \$50,000.

```
SELECT JOBCODE, MAX(SALARY), MIN(SALARY)
FROM CA.TEMPL
GROUP BY JOBCODE
HAVING COUNT(*) > 1 AND MAX(SALARY) < 50000
```

### Example 3

Show all rows of the employee-to-project-activity table TEMPRAC for Department E11, as determined by the employee table TEMPL.

```
SELECT *
FROM CA.TEMPRAC
WHERE EMPNO IN (SELECT EMPNO
                FROM CA.TEMPL
                WHERE WORKDEPT = 'E11')
```

### Example 4

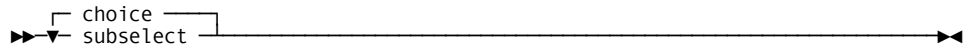
Show the department number and maximum departmental salary for all departments whose maximum salary is less than the average salary for all employees. The information is in the employee table TEMPL. In this example, the subselect would be executed only once.

```
SELECT WORKDEPT, MAX(SALARY)
FROM CA.TEMPL
GROUP BY WORKDEPT
HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                     FROM CA.TEMPL)
```

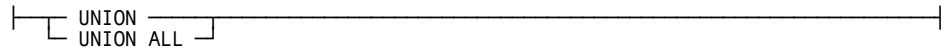
## Full-Select Statement

A full-select statement specifies a result table. If UNION is not used, the result of the full-select is the result of the specified subselect.

Following is the syntax diagram for the full-select statement:



*Expansion of Where choice is as follows*



## Description

### UNION

Derives a result table by combining two other result tables. The set of rows in the UNION of result tables R1 and R2 is the set of rows in either R1 or R2, with redundant duplicate rows eliminated. Each row of the UNION table is either a row from R1 or a row from R2.

The columns of the result table are not named.

### UNION ALL

As in UNION, derives a result table by combining two other result tables. The set of rows in the UNION ALL of result tables R1 and R2 is the set of rows in either R1 or R2, but UNION ALL specifies not to eliminate duplicate rows when deriving the result table. Each row of the UNION ALL table is either a row from R1 or a row from R2.

The columns of the result table are not named.

### *subselect*

Specify a subselect. For more information about the subselect see Subselect.

**Important!** If more than two subselects are used, you may not mix UNION with UNION ALL.

## Duplicate Rows

Two rows are duplicates of one another only if each value in the first row is equal to the corresponding value of the second row.

A UNION eliminates all but one row of each set of duplicates. The number of rows in the UNION table is the sum of the number of rows in R1 and R2, less the number of duplicates eliminated.

If ALL is specified, duplicate rows are not eliminated.



## Rules for Columns

Result tables R1 and R2 must have the same number of columns.

When UNION is performed, the corresponding columns of the SELECT lists need to be compatible data types.

### Example 1

Show all the rows from the EMP table.

```
SELECT *
FROM EMP
```

### Example 2

List the employee numbers of all employees:

- Whose department number begins with D or,
- Who are assigned to projects whose project number begins with AD.

```
SELECT EMPNO
FROM EMP
WHERE WORKDEPT LIKE 'D%'

UNION

SELECT EMPNO
FROM ASSIGNTBL
WHERE PROJNO LIKE 'AD%'
```

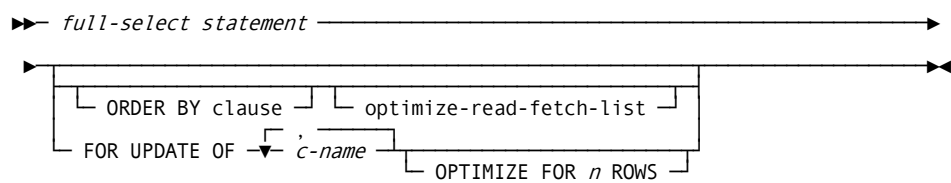
## Select-Statement

The select-statement is the form of a query that you can issue in interactive SQL (using CA Datacom Datadictionary or CA Dataquery online) or in static SQL (embedded in a DECLARE CURSOR statement in a preprocessed batch program), but not in dynamic SQL. The result table returned by a select-statement is the result of the full-select statement.

Following is the syntax diagram for the select-statement:

**Note:** For the syntax of the full-select statement, see [Full-Select Statement](#) (see page 776).

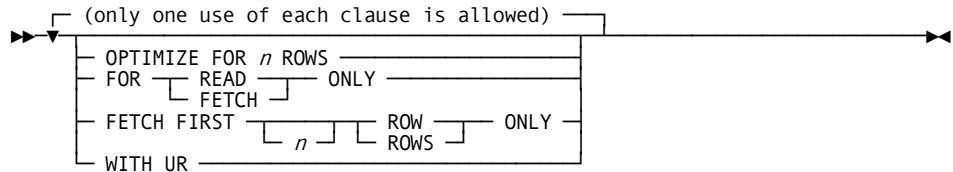
FOR UPDATE OF is a CA Datacom/DB extension.



**Note:** OPTIMIZE FOR *n* ROWS is a CA Datacom/DB extension.

FOR READ/FETCH ONLY is a CA Datacom/DB extension.

FETCH FIRST is a CA Datacom/DB extension.



## Description

### ORDER BY *clause*

You can specify the order for rows in the result table. For more information, see [ORDER BY Clause](#) (see page 779).

### FOR UPDATE OF *c-name*

This CA Datacom/DB extension is provided for compatibility with other SQL implementations. The functionality of update indication is accomplished by other means in CA Datacom/DB. The columns must belong to the table or view that is named in the FROM clause of the SELECT statement. Commas must separate the column names (*c-name* in the diagram).

### OPTIMIZE FOR *n* ROWS

This CA Datacom/DB extension tells the optimizer how many rows are normally fetched before the cursor is closed. This is useful when there is a choice between:

- A key that builds a temporary table and gives a lower cost if all rows are fetched, and
- A key that does *not* build a temporary table and takes longer if all rows are fetched, but takes less time when only a few rows are fetched

**Note:** The number *n* is *not* a limit (as it is, for example, in FOR FIRST *n*). The complete set of rows can be fetched. The number *n* in this clause is merely input to the optimizer.

### FOR READ/FETCH ONLY

This clause is provided for compatibility with other SQL implementations. If you use it and then try to do an update or delete on the current cursor, you receive an SQL return code -130 CURSOR NOT UPDATABLE (SQLSTATE 24S05).

**NOTE:** FOR FETCH ONLY does *not* force locking or have any effect on locking in any way. It only prevents the cursor from having an UPDATE/DELETE WHERE CURRENT OF cursName.

**FETCH FIRST *n* ROW/ROWS ONLY**

The FETCH FIRST clause allows you to control how many rows a query retrieves. This clause allows you to limit the number of fetches that are done. FETCH FIRST can therefore be used to prevent runaway queries.

**Note:** Queries that require the resulting set to be sorted have to perform the sort on the entire resulting set. For example, if a non-indexed column is being ordered by the use of an ORDER BY clause, to return the correct *n* rows the entire resulting set would need to be generated and sorted. In this case, the savings would be minimal even though CA Datacom/DB would prevent an excessive number of FETCHes from occurring.

The *n* can be optionally used to specify the number of rows to be fetched. When a specified number of rows have been fetched, a SQL return code number +100 NO ROW FOUND (SQLSTATE 02000) is received. If *n* is not specified, the resulting set is limited to one row.

Either ROW or ROWS must be specified.

ONLY must be specified. ONLY is provided for compatibility with other SQL implementations.

**QUERYNO *nnnnn***

Use this optional clause to identify a query in various reports. The supplied value is stored in the Source Cache Entry Dynamic System Table SC\_ENTRY. If this option is not used, a default unique negative value is used in SC\_ENTRY.

Valid values are any positive integer.

**Note:** CA Datacom does not check for a unique value, but because the intention is to uniquely identify the query, we recommend that you use a unique value.

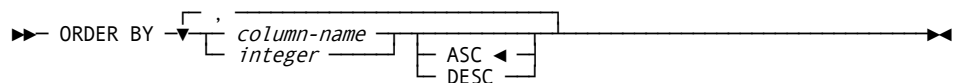
**WITH UR**

Specifies "U" transaction isolation level, which does not acquire a lock on rows read by the cursor. This option overrides the isolation level set at the plan or CA Datacom Server level.

## ORDER BY Clause

The ORDER BY clause is a component of a select-statement.

Following is the syntax diagram for the ORDER BY clause:



## Description

### **ORDER BY**

Puts the rows of the result table in order by the values of the columns you identify. If you identify more than one column, the rows are ordered first by the values of the column you identify first, then by the values of the column you identify second, and so on. If you do not specify the ORDER BY clause, the rows of the result table have an arbitrary order.

### ***column-name***

Each column-name specified must unambiguously identify a column of the result table. The sum of the lengths of the columns must be less than 32720. The column names must be separated by commas.

### ***integer***

The integer *n* identifies the *n*th column of the result table. Each integer specified must be greater than 0 and not greater than the number of columns in the result table. The integers must be separated by commas.

A named column can be identified by an integer or a column-name.

An unnamed column must be identified by an integer. A column is unnamed if it is derived from:

- A literal
- An arithmetic expression
- A function

If the full-select statement includes a UNION operator, every column of the result table is unnamed.

### **ASC**

Places the values of the column in ascending order. ASC is the default order.

### **DESC**

Places the values of the column in descending order.

Ordering is performed in accordance with the comparison rules. The null value is higher than all other values. If your ordering specification does not determine a complete ordering, rows with duplicate values of the last identified column have an arbitrary order.

## Processing

A cursor defined with an ORDER BY clause cannot be used for update.

## Examples

In this example, the ORDER BY clause is used to order the results of the SELECT. The SELECT retrieves the number and name of all employees hired before 1980 in order of seniority. HIREDATE is in the form yymmdd.

```
SELECT EMPNO, LASTNAME, FIRSTNAME, HIREDATE
FROM EMP
WHERE HIREDATE < 800000
ORDER BY HIREDATE
```

## Example

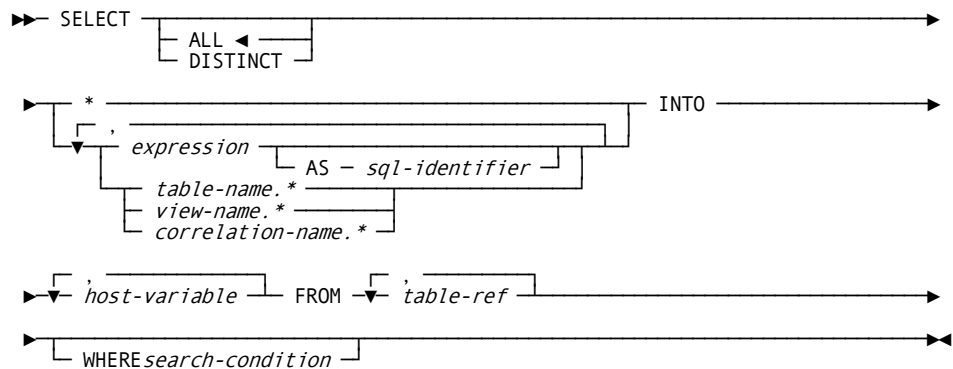
This example shows an SQL query, using an ORDER BY clause, that runs against system tables (SYS authorization ID) to display keys, where TBL and FLD are correlation names.

```
SELECT DISTINCT TBL.TABLE_SQLNAME,
FLD.DBID,
FLD.KEY_NAME,
FLD.OCCURRENCE,
FLD.KEY_FIELD_SEQ,
FLD.FIELD_LENGTH,
FLD.FIELD_UNIQUE
FROM SYS.DIR_TABLE TBL,
SYS.DIR_KEY_FIELD FLD
WHERE (TBL.TABLE_NAME = FLD.TABLE_NAME)
AND
(TBL.TABLE_SQLNAME = 'PSN_AST' AND TBL.STATUS = 'P')
ORDER BY FLD.DBID, FLD.KEY_NAME, FLD.KEY_FIELD_SEQ;
```

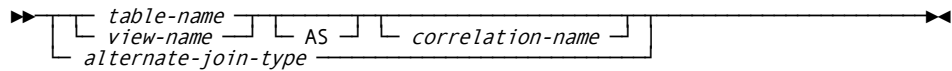
## Select-Into Statement

The select-into statement produces a result table consisting of at most, one row, and assigns the values in that row to host variables. If the table is empty, the statement assigns +100 to SQLCODE and does not assign values to the host variables.

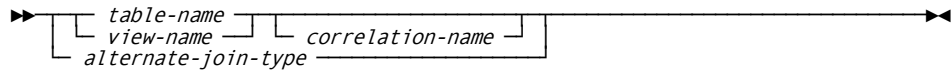
Following is the syntax diagram for the select-into statement:



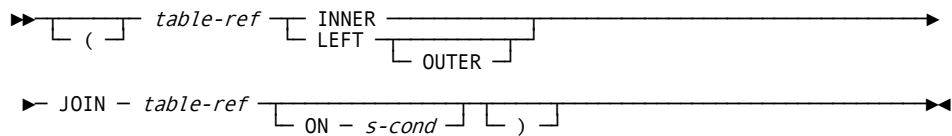
The table-ref shown in the syntax box immediately preceding the following one has syntax as follows:



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 for a LEFT JOIN 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.

## Description

### SELECT Clause

#### SELECT

Produces a final result table by selecting only the columns indicated by the select list from R, where R is the result of the previous clause.

**Note:** We do not impose an arbitrary limit on the number of columns you can select in a query. We are limited only by environmental factors. Some of these factors are the size of your work area as specified by the *size* parameter of the TASKS Multi-User startup option (see the *CA Datacom/DB Database and System Administration Guide*), your column sizes, and limits placed by other products such as the CA Datacom Server.

#### ALL

Retains all rows of the final result table and does not eliminate redundant duplicates. ALL is the default.

**DISTINCT**

Eliminates all but one of each set of duplicate rows of the final result table.

DISTINCT must not be used more than once in a select-into statement.

\*

The asterisk (\*) represents a list of names that identify the columns of R. The first name in the list identifies the first column of R, the second name identifies the second column of R, and so on.

The list of names is established when the program containing the SELECT clause is prepared.

**Important!** SELECT \* and SELECT table.\* (see the following) are useful when selecting from tables or views in an interactive environment, especially when the names of the columns are not known. However, embedding SELECT \* or SELECT table.\* in an application program may cause unexpected results if the definition of the table is ever altered. For example, when a column is dropped from a table definition, all statements which reference the table are automatically rebound as they are executed. The rebound forms of the statements reflect the new set of columns. The application program, however, still expects the original set of columns, and the result table returned to the program no longer matches the FETCH statement's host variables. Views which are referenced by application programs should not include SELECT \* or SELECT table.\* for the same reason.

***name.\****

The asterisk (\*) represents a list of names that identify the columns of R. The **name** can be a table-name, view-name, or correlation-name, and must designate a table-name or view-name in the FROM clause. The first name in the list identifies the first column of R, the second name the second column, and so on. The list is established at preparation time and does not represent any columns that have been added later. The names must be separated by commas.

***expression***

Commonly, the expressions used in a SELECT statement include column names. Each column name used in the select list must unambiguously identify a column of R. For more information about expressions, see Expressions. Multiple expressions must be separated by commas.

**AS**

*(Optional)* The AS clause, used after an expression, can be used to give a name to an expression or to give a column reference another name that is returned to the user in the SQLDA. The term *sql-identifier* represents either an ordinary or delimited SQL identifier (see Identifiers). The word AS itself is optional.

An AS can also be used before the *correlation-name* in the *table-ref* part of a FROM clause.

An AS clause is used in the following example to give the name PROPOSED\_SALARY to the expression:

```
SELECT EMP.NAME, SALARY, SALARY * 1.1
       AS PROPOSED_SALARY
FROM EMP;
```

The number of columns in the result of SELECT is the same as the number of expressions in the operational form of the select list, that is to say, the list established at preparation time. The result of a subquery must be a single column unless the subquery is used in the EXISTS predicate.

## Applying the Select List

The select list must either include no functions, or be entirely a list of functions.

If the select includes no functions, then the select list is applied to each row of R, and the result contains as many rows as there are rows in R.

If the select list is a list of functions, then R is the source of the arguments of the functions, and the result of applying the select list is one row.

The nth column of the result contains the values specified by applying the nth expression in the operational form of the select list.

A result column derived from a column name acquires the unqualified name of that column. All other result columns have no names.

Each column of the result of SELECT acquires a data type from the expression from which it is derived.



---

## INTO Clause

Introduces a list of host variables.

*host-variable*

### **INTO**

Must name a variable that is described in the program in accordance with the rules for declaring host variables. The names must be separated by commas.

The first value in the result row is assigned to the first variable in the list, the second value to the second variable, and so on. If the number of values is not the same as the number of variables, the value W is assigned to SQLCA-WARNING(4).

The data type of a variable must be compatible with the value assigned to it. If the value is numeric, the variable must have the capacity to represent the integral part of the value. If the value is null, an indicator variable must be specified.

Each assignment to a variable is made according to the rules described in [Basic Operations \(Assignment and Comparison\)](#) (see page 501). Assignments are made in sequence through the list. If an assignment error occurs for some variable, no value is assigned to that variable or later variables. Any values that have already been assigned to variables remain assigned.

## FROM Clause

### **FROM table-ref**

Where table-ref can be a table-name, view-name, or alternate-join-type. If table-ref is a table-name or view-name, it names:

- A single table or view.
- Several tables and/or views to produce an intermediate result table.

If table-ref is an alternate-join-type, see the information about joins in Left Outer Joins.

You can reference up to 20 tables in a FROM clause of a query when you are performing a join. For example, if a view is based on five tables, you can name that view in the FROM clause and up to 15 other tables. The names must be separated by commas.

The intermediate result table contains all possible combinations of the rows of the named tables or views. Each row of the result is a row from the first table or view concatenated with a row from the second table or view, concatenated in turn with a row from the third, and so on. The number of rows in the result is the product of the number of rows in all the named tables or views.

The list of names in the FROM clause must conform to these rules:

1. Each table-name and view-name must name a table or view described in the CA Datacom Datadictionary.
2. The FROM clause of a select-into statement must not identify a view that includes a GROUP BY or HAVING clause.
3. If any view named contains a column function, that view-name must be the only name in the FROM clause, that is to say, you cannot have a join in this case.

***correlation-name***

The 1- to 18-byte *correlation-name* applies to the table or view named by the immediately preceding table-name or view-name. The correlation name can be used elsewhere in the statement to designate that table or view. Use a blank to separate the correlation name from the table-name or view-name.

## WHERE Clause

**WHERE *search-condition***

Produces an intermediate result table by applying the search-condition to each row of table R, which is the result of the FROM clause. The result table contains the rows of R for which the search condition is true.

**Note:** If using alternate-join-type, see the additional information about WHERE clauses in WHERE Clause. Also, in addition to the information about search conditions that follows, see the additional alternate-join-type search condition.

The search-condition describes a search condition that conforms to the following rules:

1. The condition is formed (see [Search Conditions](#) (see page 593)).
2. Each column-name in the search-condition either unambiguously identifies a column of R, or is a correlated reference. A correlated reference is allowed only in a subquery.
3. A column-name in the search-condition does not identify a column that is derived from a function or a grouping column. A column of a view can be derived from a function or a grouping column.
4. The search condition does not include a function unless the argument of the function is a correlated reference. This is only possible in a subquery of a HAVING clause.

Any subquery in the search-condition is effectively executed for each row of R and the results used in the application of the search-condition to the given row of R. A subquery is actually executed for each row of R only if it includes a correlated reference to a column of R.

## Example 1.

Put the row for employee 528671, from the employee table, into the specified host variables.

```
EXEC SQL
  SELECT *
  INTO :EMPNO, :FNAME, :MI, :LNAME, :DEPTNO,
  :HIREDATE, :SALARY
  FROM CA.TEMPL
  WHERE EMPNO = '528671'
END-EXEC
```

## Example 2.

Put the maximum salary in the employee table into the host variable MAXSALARY.

```
EXEC SQL
  SELECT MAX(SALARY)
  INTO :MAXSALARY
  FROM CA.TEMPL
END-EXEC
```

## SET CURRENT SQLID

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
SET CURRENT SQLID		YES	

**Note:** YES indicates a valid execution method for this statement.

The SET CURRENT SQLID statement can be prepared and executed dynamically. SET CURRENT SQLID allows the CURRENT SQLID special register to be changed to any authorization ID. The value of the CURRENT SQLID special register is the current SQL authorization ID (in this case actually a schema ID). Using SET CURRENT SQLID to change the CURRENT SQLID therefore allows you to qualify any unqualified table or view names in dynamically executed SQL statements. After you have changed the CURRENT SQLID special register with SET CURRENT SQLID, the value you specified remains in effect until either another SET CURRENT SQLID statement is executed or your application process terminates.

**Note:** In a Single User environment, the CURRENT SQLID affects all plans executing under an application process.

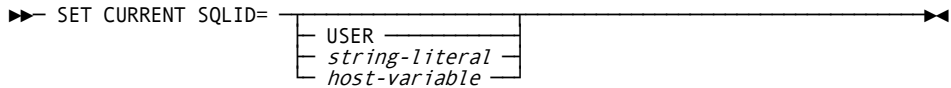
There is no authorization checking required to execute the SET CURRENT SQLID statement. The name checked for privileges when a CREATE, GRANT or REVOKE statement is dynamically prepared is the value of the accessor ID special register, not the value of CURRENT SQLID. The accessor ID becomes the owner of tables, views and indexes created by dynamic SQL statements. The value of the accessor ID special register may not be changed.

The CURRENT SQLID special register is initially set to the same value as the USER special register, which is the authorization ID of the plan that is executing. This initial value is also known as the *primary* SQL authorization ID. The USER register retains the original value even when the CURRENT SQLID is changed.

**Note:** In interactive SQL products such as CA Dataquery or CA Datacom Server, plans are generated internally, but there is always a way of specifying what authorization ID the generated plans have.

If Dynamic Plan Selection is used, the AUTHID provided in the User Requirements Table (URT) parmameter or CA Datacom CICS Services table is used to specify the authorization ID of the plan, which corresponds to the USER special register. That authorization ID is used to qualify any unqualified table or view names in the non-dynamically prepared statements in the plan. Do not confuse this with the CURRENT SQLID, which only affects dynamically prepared statements.

Following is the syntax diagram for the SET CURRENT SQLID statement:



## Description

### USER

Specifying USER means you want the CURRENT SQLID special register set to the authorization ID of the currently executing plan.

### *string-literal or host-variable*

Use a string literal or host variable if you want to specify a value for the CURRENT SQLID special register.

If the value you specify is not a valid authorization ID that was created by using the CREATE SCHEMA statement, an error results if the authorization ID is later used to qualify a table or view name in a dynamic SQL statement.

## Example

Change the current SQLID special register to contain the contents of host variable :USERSQLID.

```
EXEC SQL
    SET CURRENT SQLID=:USERSQLID
END-EXEC
```

## UPDATE

This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
UPDATE (positioned)		YES	
UPDATE (searched)	YES	YES	YES

**Note:** YES indicates a valid execution method for this statement. To learn about using SQL keywords in CA Dataquery, see the *CA Dataquery User Guide*. For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

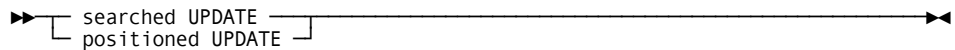
The UPDATE statement updates the values of specified columns in rows of a table or view. Updating a row of a view updates a row of the base table from which the view is derived.

You must specify ISOLEVEL=C (isolation level C) in the Preprocessor options when using the UPDATE statement.

**Note:** When the NOMAINT option of the CA Datacom/DB Utility (DBUTLTY) ACCESS function is in force, an UPDATE statement receives a CA Datacom/DB return code 94(87), where 87 is a decimal internal return code (hex 57) that tells you no maintenance statements are allowed while NOMAINT is in force.

Also note that constraints that existed in versions before r10 can act differently in Release 10.0 and above because, in Release 10.0 and above, columns containing a NULL rather than a value do *not* cause a CHECK constraint to be violated. Constraints are considered, in Release 10.0, to be satisfied unless the predicates evaluate explicitly to FALSE. That is, CHECK constraints whose predicates evaluate to UNKNOWN rather than TRUE or FALSE are considered, starting with Release 10.0, to have been satisfied. Therefore, INSERT and UPDATE statements, from versions before r10, that resulted in constrained columns being nulled are, when used in Release 10.0 and above, successful for the first time.

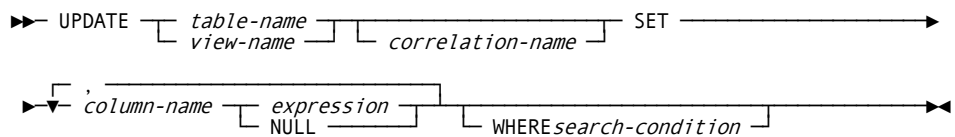
Following is the syntax diagram for an UPDATE statement:



**Note:** The positioned UPDATE is not used by CA Dataquery.

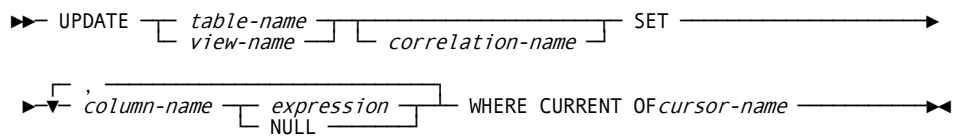
## Searched UPDATE

Following is the syntax diagram for the searched UPDATE statement:



## Positioned UPDATE

Following is the syntax diagram for the positioned UPDATE statement that you use with a cursor:



## Description

### ***table-name or view-name***

Specify the name of the table or view you want to update. You must name a table or view described in the CA Datacom Datadictionary, but not a CA Datacom Datadictionary table, a view of a CA Datacom Datadictionary table, or a read-only view.

### ***correlation-name***

You can specify a 1- to 18-byte *correlation-name* (correlation-name) to be used within the search-condition to designate the table or view. Also see [Correlation Names](#) (see page 515) and [SQL Index Binding](#) (see page 516).

### **SET**

Introduces a list of column names and values.

### ***column-name***

Specify the names of the columns you want to update. The names must be separated by commas. You must name a column of the table or view you specified. Do not specify the name of the same column more than once and do not specify a column of any of the following types:

- A view column that is derived from a literal.
- A view column that is derived from an arithmetic expression.

### ***expression***

Tells the new value of the column. The expression cannot include a function. See [Expressions](#) (see page 527) for the expression's syntax diagram.

A column-name in an expression must name a column of the named table or view. For each row that is updated, the value of the column in the expression is the value of the column in the row *before* the row is updated. The following table shows a SET clause, the value of the columns in the row (before) and the resulting value (after) for each column:

SET clause	Value of X Before	Value of Y Before	Value of X After	Value of Y After
SET X = 10, Y = X + 1	1 3	1 5	1 10	1 4

### **NULL**

Use the NULL keyword to specify that the value for the column is to be set to a null value.

### **WHERE**

Introduces a condition that tells what rows are updated. You can omit the clause, give a search condition, or name a cursor. If you omit the clause, all rows of the table or view are updated.

**search-condition**

The search-condition is applied to each row of the table or view and the updated rows are those for which the result of the search-condition is true. See [Search Conditions](#) (see page 593) for the search-condition syntax diagram.

Each column-name in the search condition must name a column of the table or view, and the table or view must not be referenced in the FROM clause of any subselect in the search condition.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a row, and the results used in applying the search condition. In actuality, the subquery is executed for each row only if it contains a correlated reference to a column of the table or view.

**CURRENT OF cursor-name**

Specify the name of a cursor that is defined in a DECLARE CURSOR statement of your program. The DECLARE CURSOR statement must appear in your program before the UPDATE statement.

The table or view named must also be named in the FROM clause of the SELECT statement of the cursor, and the result table of the cursor must not be read-only.

When the UPDATE statement is executed, the cursor must be positioned on a row. That row is the one that is updated.

## Processing

Update values are assigned to columns in accordance with the following assignment rules:

<b>If update value is:</b>	<b>The column must be:</b>
A number	A numeric column with the capacity to represent the integral part of the number
A character string	A character string column with a length attribute that is not less than the length of the string

The updated row must conform to any constraint imposed on the table (or on the base table of the view) by any unique index on an update column. If an update value violates any of those constraints, or if any other error occurs during the execution of the UPDATE statement, no rows are updated. If an error occurs during the execution of UPDATE that makes the position of a cursor unpredictable, the cursor is closed. Rows can be changed so that they no longer conform to the definition of the view. Such rows are updated in the base table of the view and no longer appear in the view.

Unless appropriate locks already exist, one or more exclusive locks are acquired by the execution of a successful UPDATE statement. Until the locks are released, an updated row can be accessed only by the unit of recovery that performed the update.



### Example 1

For employee 000190, change the employee's telephone number in the TEMPL table.

```
EXEC SQL
    UPDATE CA.TEMPL
    SET PHONENO = '3565'
    WHERE EMPNO = '000190'
END-EXEC
```

### Example 2

Increase the job code by 10 members in Department D11.

```
EXEC SQL
    UPDATE CA.TEMPL
    SET JOBCODE = JOBCODE + 10
    WHERE WORKDEPT = 'D11'
END-EXEC
```

### Example 3

This example assumes the system for assigning manager numbers has been modified due to changes at the division level. To locate the manager numbers which must be updated, this example uses a SELECT in a DECLARE CURSOR statement. When the result table is obtained and the cursor is positioned, an UPDATE statement, using the WHERE CURRENT OF clause, updates the value for MGRNO.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT DEPTNO, DEPTNAME, MGRNO
        FROM DEPTTBL
        WHERE ADMDEPT = 'A0'
END-EXEC
EXEC SQL
    OPEN C1
END-EXEC
FETCH-LOOP.
IF SQLCODE = 0
    EXEC SQL
        FETCH C1 INTO :DNUM, :DNAME, :MNUM
    END EXEC.
    EXEC SQL
        UPDATE DEPTTBL
        SET MGRNO = MGRNO + 100000
        WHERE CURRENT OF C1
    END-EXEC.

    GO TO FETCH LOOP.
EXEC SQL
    CLOSE C1
END-EXEC
```

# WHENEVER

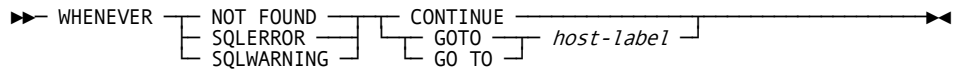
This SQL statement can be executed in the following ways:	Through the CA Datacom Datadictionary Interactive SQL Service Facility ( <i>interactive</i> )	In an application program prepared using a CA Datacom/DB SQL Preprocessor ( <i>embedded</i> )	By using CA Dataquery ( <i>SQL &amp; Batch Modes</i> )
WHENEVER		YES	

**Note:** YES indicates a valid execution method for this statement. For information about the access rights required to execute this statement, see the *CA Datacom/DB Database and System Administration Guide*.

The WHENEVER statement identifies the statement that is to be executed next if execution of the latest SQL statement produces a specified condition.

Following is the syntax diagram for the WHENEVER statement:

**Note:** SQLWARNING is a CA Datacom/DB extension.



## Description

Use the following clauses to identify the type of exception condition.

### NOT FOUND

Identifies any condition that results in an SQL return code of +100.

### SQLERROR

Identifies any condition that results in a negative SQL return code.

### SQLWARNING

This CA Datacom/DB extension allows you to identify any condition that results in a warning condition (SQLCA-WARNING is W) or that results in a positive SQL return code other than +100.

Use the following clauses to specify the statement to be executed when the identified type of exception condition exists.

#### **CONTINUE**

Causes the program to continue execution with the next sequential statement of the source program.

#### **GO TO or GOTO**

Causes transfer of control to the statement identified by host-label.

#### ***host-label***

You can specify a single token, optionally preceded by a colon. The form of the token depends on the host language. For example, in COBOL it can be a section-name or an unqualified paragraph-name.

## Processing

The types of WHENEVER statements are:

<b>ANSI</b>	<b>Extended Mode</b>
WHENEVER NOT FOUND WHENEVER SQLERROR	WHENEVER NOT FOUND WHENEVER SQLERROR WHENEVER SQLWARNING

Every executable SQL statement in a program is within the scope of one implicit or explicit WHENEVER statement of each type. The scope of a WHENEVER statement is related to the listing sequence of the statements in the program, not their execution sequence.

The scope of an exception declaration:

1. Begins with the placement of the exception declaration in the source, and
2. Continues until either another exception declaration in the source or the end of the source.

An SQL statement is within the scope of the last WHENEVER statement of each type specified before that SQL statement in the source program. SQL statements occurring in the source before an exception declaration are not affected by it. Any SQL statements executed that are not under control of an exception declaration default to CONTINUE. If an exception declaration is not provided, the recommended practice is that your program include code to check the SQLCODE value in the SQL Communication Area (SQLCA) immediately after each executable SQL statement.

## Example

The following identifies the exception conditions and the next statement to be executed in the order listed:

1. If an error is produced, go to HANDLERR.
2. If a warning code is produced, continue with the normal flow of the program.
3. If no results are found, go to ENDDATA.

## WHILE Statement

For details about this statement, see [WHILE Statement](#) (see page 675).

**Note:** This statement is executed from within the Compound Statement (see [Compound Statement](#) (see page 642)).

# Appendix A: SQL Query Optimization Messages

---

You can tune the performance of SQL queries by using the SQL Query Optimization Messages to determine:

- What processing steps the data access plan contains,
- Why those processing steps were selected, and
- The estimated and actual costs of those processing steps.

There are two main types of Optimization messages:

- Bind Time (see [Bind-time Messages](#) (see page 798)):

**Summary:**

Join order and method, sort required, total estimated cost.

**Detail:**

Detail estimates for each join order/method combination.

- Execution Time (see [Execution-Time Messages](#) (see page 812)):

**Summary:**

Rows read from index and data, rows qualified, rows sorted, and so on, during the time the plan was open.

**Detail:**

Same data as summary, but for each execution of a statement.

## Message Table (SYSADM.SYSMSG)

Optimization messages are written to table SYSADM.SYSMSG. They are deleted when the plan is rebound or deleted.

This table was created as:

```
CREATE TABLE SYSADM.SYSMSG
(AUTHID CHAR(18) NOT NULL,
 PLANNAME CHAR(18) NOT NULL,
 STMTID INTEGER NOT NULL,
 SEQNBR SMALLINT NOT NULL,
 MSG CHAR(80) NOT NULL,
 PRIMARY KEY (AUTHID, PLANNAME, STMTID, SEQNBR))
IN SYSMSG_AREA;
```

## Requesting Messages

You specify that SQL Query Optimization Messages be generated by either using the MSG= plan option (see CA Datacom/DB SQL Preprocessors) or by using the COMM function of CA Datacom/DB Utility (DBUTLTY) as follows.

For summary level:

```
COMM OPTION=ALTER, TRACE=TRACEMSG, JOBNAME=xxxxxxx
```

For detail level, also turn on:

```
COMM OPTION=ALTER, TRACE=TRACEDETAIL, JOBNAME=xxxxxxx
```

When triggered by CA Datacom/DB Utility (DBUTLTY) traces, the messages are written both to SYSADM.SYSMSG and to the PXX. For more information on using the CA Datacom/DB Utility (DBUTLTY), see the *CA Datacom/DB DBUTLTY Reference Guide*.

**Note:** CA Dataquery deletes the messages from the previous query when the next query is executed if the plan option to generate messages is not turned off. Therefore, to save the messages turn off the message option and run:

```
SELECT * FROM SYSADM.SYSMSG
```

You can optionally use the WHERE clause to read only rows for the desired plan. After the SELECT statement has been run, use the DQRY STORE command to store the result in a different table. Alternately, before preparing another query (and without signing off DQRY) use any other method, such as DDOL, to query the SYSADM.SYSMSG table.

## Bind-time Messages

Cost estimates used in bind time messages are based on the number of index and data blocks that must be accessed.

## Bind-time Summary Messages

Summary-level messages contain:

- Reference Information (source statement and indexes)
- Sort optimization
- Data Access Plan (join order and method, sorts required, total estimated cost)

Detail-level messages contain cost estimates for each possible combination of join order and method. If you think another join order or method should have a lower cost, you can see the estimates used by the optimizer.

## Reference Information

This section identifies the statement and subselect or subquery being optimized and contains a copy of the SQL source statement and index definitions for referenced tables.

### *Headings and Source Statement*

Bind-time messages begin by identifying the statement being bound and listing the SQL statement:

```
*** Plan:authId.planName, Stmt:nnnnnnn DT:yyyy.-mm-dd hh:mm:ss
xxxxxxxxxxxxxxxxxxxxxxxxxxxx source statement xxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

One of the following messages identifies the subselect or subquery:

```
***** BIND MESSAGES FOR SUBSELECT nn *****
```

```
***** BIND MESSAGES FOR SUBQUERY LEVEL nn NUMBER nn OF SUBSELECT nn *****
```

### **SUBSELECT**

Subselects are numbered in the order they appear in the SQL statement. Unless you have a UNION, there is only one subselect.

### **LEVEL**

Level 1 is a subquery of a subselect, level 2 a subquery of a level 1 subquery, and so on.

### **NUMBER**

Multiple subqueries in the same search condition are numbered in the order in which they appear in the SQL statement.

### *Index Definitions*

Index definitions are listed as reference information for each base table referenced in the query.

### **Table Level**

```
INDEX DEFINITIONS FOR: ttt/ddd authId.tableName correlationName
KEYS=nn, IDXLVLS=n, LN=nnnnn, ROWS/BLK=nnn ROWS=nnnnnnnnn
*** WARNING: INDEX CARDINALITY STATISTICS NOT COMPUTED ***
```

### **ttt/ddd**

DATACOM 3-character name and database ID.

### **KEYS**

Number of keys defined for the table.

**IDXLVLS**

Number of levels in the index.

**LN**

Number of bytes in a row.

**ROWS/BLK**

Number of rows that can fit in a data block (actual value may be higher for compressed areas).

**ROWS**

Number of rows in the table (as reported in the Directory (CXX)).

**CARDINALITY NOT COMPUTED**

Join optimization is very dependent on index cardinality. To compute index cardinality, use the CA Datacom/DB Utility (DBUTLTY) to either execute the RETIX or LOAD function, or use:

```
REPORT AREA=IXX,DBID=nnn,TYPE=G,UPDATE=YES
```

**Key Level**

```
KEY xxxxx id=nnn FLG=hh hh FLDS=nn DXX=nnnn BLKCHG=nnnn ROWS=nnnnnn
```

**KEY**

CA Datacom/DB five-character key name.

**ID**

CA Datacom/DB key ID.

**FLG**

Key attributes (as hex dump of bit flags):

**Note:** Any value found (on a report) that is unobtainable using the values below is for internal use only.

**x'80'**

NATIVE SEQUENCE KEY

**x'40'**

NIL-INCL-KEY

**x'10'**

MASTER KEY



**x'08'**

DUPLICATE KEY ID IN THIS FILE

**x'04'**

DIFFERENT TOTAL KEY LENGTHS

**x'01'**

UNIQUE KEY

**x'20'**

KEY HAS DATA TYPE SENSITIVE FIELDS

**x'10'**

KEY HAS DECIMAL FIELDS

**x'08'**

KEY IS RELATIVE RECORD NUMBER

**x'01'**

CBS WILL IGNORE THIS KEY

**FLDS**

Number of columns in key

**DXX**

Average number of index entries per DXX block

**BLKCHG**

Average number of data area blocks that must be accessed to read 1024 rows in sequence by this key.

**ROWS**

Number of index entries at the time when cardinality was last computed. This can be lower than ROWS at the table level if NIL-INCL-KEY is specified.

**Column Level**

OFFSET=nnnnn, LN=nnn, DIR=xxxx SENS=x CARDINALITY=nnnnnnnnn xxxxx...

**OFFSET**

Offset of column in the row, relative to zero

**LN**

Number of bytes in the column

**DIR**

Either ASC for ascending or DESC for descending direction

**SENS**

Either Y for data type sensitive, or N

**CARDINALITY**

Number of unique values for this and preceding columns

xxxxx...

SQL column name

**Sort Optimization**

Each subselect or subquery can require at most two sorts for:

- SORT1: GROUP BY
- SORT2: ORDER BY, UNION, or DISTINCT

The estimated cost of these sorts is indicated by the following messages:

ESTIMATED GROUP SORT COST= nnnnnnnnn

ESTIMATED ORDER/DISTINCT/UNION SORT COST= nnnnnnnnn

If ORDER BY and GROUP BY reference the same columns, SORT1 eliminates the need for SORT2, which is indicated by the following message:

ORDER/DISTINCT/UNION SORT SATISFIED BY GROUP BY SORT

UNION and DISTINCT always require SORT2, but if the scan index returns rows in sort sequence, then sorts for GROUP BY and ORDER BY can be eliminated.

The following messages indicate which indexes, if any, satisfy these sorts.

KEY xxxxx SATISFIES GROUP BY

KEY xxxxx SATISFIES ORDER/DISTINCT/UNION

However, if these indexes are to be used to eliminate sorts, they must:

- Index the first table in join sequence, and
- Be the same index as index merge, if index merge used.

**Data Access Plan**

The data access plan contains:

- Estimated cost
- Order and method in which tables are joined

- Scan indexes with which estimates were calculated (except for index-merge join indexes, and indexes used to eliminate sorting, different indexes may be used at execution-time due to the value of host variables or index statistics).
- Sorts required

#### *Total Estimated Cost*

Total estimated cost is given by the first message if there is a join, else by the second message.

TOTAL JOIN COST ESTIMATE = nnnnnnnnn

KEY xxxxx HAS LOWEST COST OF nnnnnnnnn FOR nnnnnnnnn ROWS

#### *Join Steps*

Messages in this section are only generated when a join exists.

The terms *outer* and *inner* table reference the first and second tables of a join, respectively. When three or more tables are joined, the first two tables are joined, and the result of that join becomes the outer table of the next join, and so on.

Predicates dependent only on a single table are called *restriction* predicates. These predicates are always applied before join conditions. The terms *inner* and *outer* table reference to conceptual intermediate tables containing only those rows of their base table for which restriction predicates are true.

#### **JOIN STEPS:**

Each join step is described in the order of execution. Different messages are issued depending on the join method used.

#### **Nested-Loop Join Method**

The nested-loop join method searches the inner table for *matching* rows for each outer table row. A matching row is a row for which the join conditions are true.

Nested-loop is used when:

- Merging is not possible.
- Has the lowest estimated cost.
- Manual optimization is specified with plan option MSG.

An example of when nested-loop has the lowest estimate cost:

- Outer table has a non-join index restriction (merge would need to read the entire unrestricted join index), or a non-join index can eliminate a sort due to GROUP BY or ORDER BY.
- Inner table has an index restricted by the join condition (the entire table does not need to be searched).  
NESTED-LOOP JOIN TBL n USING KEY xxxxx  
TO TBL n USING KEY xxxxx

**TBL**

Indicates table by its position in the FROM clause.

**KEY**

CA Datacom/DB five-character key name that has the lowest estimated cost. This may not be the key used at execution time.

**TO**

Only the first join generates both lines. The outer table of subsequent joins is the *intermediate result table* of previous joins. This intermediate result table is not materialized, that is to say, it is not physically generated as a temporary table in the TTM area.

**Index Merge Join Method**

Index merge can only be used when:

- A first join.
- There is at least one equijoin condition.
- There are no non-equijoin join conditions.
- Operands of equijoin predicates are column references (no expressions or scalar functions) but data type conversion is acceptable.
- Both tables have a *matching* index with all join columns in the same order.
- No OR has more than one table *under* it.
- There are no expressions in the search condition referencing more than one table.

An example of when index merge has the lowest cost:

- Search condition contains only equijoin predicates (no restrictions).
- Restriction predicates do not reference indexed columns.

Under these conditions index merge has the following advantages over nested-loop:

- Only one set is required for the inner table. Nested-loop requires a set for each outer table row.
- Since there is only one inner table set, it is usually larger and benefits more from pre-fetch.

```
INDEX-MERGE JOIN TBL n USING KEY xxxxx TO TBL n USING KEY xxxxx
```

#### **TBL**

Table is identified by its position in the FROM clause.

#### **KEY**

CA Datacom/DB five-character key name of index used for the merge.

#### **Sort-Merge Join Method**

The sort-merge join method has the same restrictions as index-merge except that no matching indexes are required. Instead, the tables are sorted in the sequence the matching indexes would have provided before merging begins.

These sorts build temporary tables in the TTM area.

An example of when sort-merge is faster than nested-loop:

- There is no index on the inner table restricted by the join predicates. Nested-loop must read the entire inner table for each outer table row, which is the entire base table unless there are restrictions that restrict an index scan range.
- Non-indexed restrictions exist on the inner table. Nested-loop would need to read rows rejected by the restriction for each outer table row, but sort-merge only rejects these rows once on input to the sort.

```
SORT TBL n FOR SORT-MERGE JOIN
SORT TBL n FOR SORT-MERGE JOIN
SORT-MERGE JOIN TBL n TO TBL m
```

#### **TBL**

Table is identified by its position in the FROM clause.

#### *Sorts Required*

The following messages indicate if a sort is eliminated by using an index:

```
KEY xxxxx USED FOR GROUPING
KEY xxxxx USED FOR ORDERING
```

If a sort is required, its reasons are given in the following messages:

```
GROUP BY ..... SRT REASONS= xxxxxxxx
ORDER BY/DISTINCT SRT REASONS= xxxxxxxx
```

**CBS ORDER**

The Compound Boolean Selection Facility cannot return rows in sort sequence because no index satisfies sort sequence.

**EXPR/FUNC**

ORDER BY refers to a SELECT list column that is an expression or function.

**MULTI-TBL**

ORDER BY or GROUP BY refers to columns in multiple tables.

**JOIN**

Index-merge join method is used. Its index does not satisfy ordering requirements.

**DISTINCT**

DISTINCT specified.

**UNION**

UNION (without ALL) specified.

*Predicates Evaluated in SQL-Subsystem*

The following message indicates how many predicates are evaluated by the SQL subsystem:

```
***** nn PREDICATES EVALUATED BY SQL SUBSYSTEM IN JOIN STEP nn *****
```

These predicates cannot restrict index scan range, or be evaluated from the index; so, they cause data scanning.

If the data is remote, rows may be transferred to the requesting node only to be rejected by these predicates.

The following predicates are evaluated by the SQL subsystem:

- When a column is compared to a literal, host or register variable that has a greater length, scale, or whole number digits.
- LIKE, other than pattern xxx%.
- IN list.
- Comparing two columns in the same table with different data types, precision, scale, (although different data type modifiers type-numeric, sign, and nullable are permitted).
- Correlated or quantified subquery.
- Predicates with operands that are expressions or scalar functions (except concatenation of contiguous not nullable character columns, and arithmetic with only literals).
- Predicates *under* an OR containing reference to more than one table.

- Index or sort merge equijoin predicates.
- Predicates whose operands make no reference to a column.
- All predicates in the HAVING search condition.

You can dump the actual predicates by using the CA Datacom/DB Utility (DBUTLTY) request:

```
COMM OPTION=ALTER, TRACE=TRACECRS, JOBNAME=xxxxxxxx
```

If you cannot convert a predicate into a type that restricts index scan range or be evaluated from the index, you only improve efficiency slightly or make it worse by converting it to a predicate that the Compound Boolean Selection Facility (CBS) evaluates.

For example, converting the predicate:

```
"COL1 IN(&HOST1., &HOST2.)"
```

to

```
"COL1 = :HOST1 OR COL1 = :HOST2"
```

would not help much unless COL1 is in the scan index. Performance could be worse if there are many other predicates ANDed with this predicate that are evaluated by Compound Boolean Selection. This is because these other predicates are repeated for each IN list entry to be in the CBS-required disjunctive normal form.

## Bind-time Detail Messages

The detail-level bind-time messages show the cost estimates for each combination of join order and method.

Join optimization messages are produced in the following way. *Each level of indentation indicates a loop at that level:*

- **Restriction Costs:** Cost of each table as if it is the only table. This estimate is used when the table is the first table of a candidate join sequence.
- **Join Order:** All possible orders are estimated, until the cost is greater than an order already computed.
  - **Join Method:** For each join step, the cost of each join method is estimated.
  - **Scan Index:** For nested-loop, the cost of using each inner table index is estimated. For index-merge, only matching indexes are considered. For sort-merge, the restriction index is used.

The join order with the lowest estimated cost is repeated at the end after the following heading:

----- LOWEST COST CANDIDATE -----

**Note:** Because of the number of rows generated, do not use this option when joining more than 6 tables.

**Restriction Costs**

TBL xxx DBID nnn RESTRICTION COSTS:

**RESTRICTION COSTS**

This is the estimated cost without join conditions, that is to say, as if this table were the outer table of the first join. Restriction cost estimates are computed once before all possible join orders are estimated.

**Join Step Detail**

CANDIDATE JOIN SEQUENCE = n,n,n...  
JOIN STEP n

**CANDIDATE JOIN SEQUENCE**

The cost of all possible join orders is estimated, until the estimated cost exceeds a previously computed estimate. The numbers reference table positions in the FROM clause.

**JOIN STEP**

The first two tables are joined in join step 1. The output of this step is joined to table three, and so on. The cost of each join step is estimated separately.

*Nested-Loop*

NESTED LOOP JOIN COSTS:

KEY xxxxx 1ST nn FLDS SELECTIVITY= .nnnnnnnnn  
nn LOW-ORDER FLDS SELECTIVITY= .nnnnnnnnnn  
DATA SELECTIVITY= .nnnnnnnnnn  
INDEX nnnnnnnnn DATA nnnnnnnnn SORT nnnnnnnnn ROWS nnnnnnnnn  
\*\*\* KEY xxxxx HAS LOWEST ESTIMATED COST OF nnnnnnnnn  
\*\*\* RESTRICTION COST =nnnnnnnnn  
\*\*\* NESTED LOOP COST =nnnnnnnnn



**SELECTIVITY**

Used to estimate the number of rows that are *filtered out*. For example, a selectivity factor of 1.0 indicates no rows rejected, 0.5, half the rows rejected, and 0.0 all rows rejected.

**1ST FLDS SELECTIVITY**

The first nn columns of the key are either restricted to a single value, or the last column may be restricted to a range of values.

This type of restriction is called a *high-order* restriction. The high-order restriction reduces the number of index entries that must be scanned.

Selectivity is based on cardinality statistics. For example, if the first three columns have a high-order restriction and the cardinality at that level is 1000, that is to say, there are 1000 different values within the first three columns of the key, then selectivity is 0.001. This means that when nested-loop is used, each outer table row is estimated to join to only 1 out of every 1000 rows in the inner table.

**LOW-ORDER FLDS SELECTIVITY**

This is the filtering effect of predicates that can be evaluated from the index, but do not restrict the range of index entries that must be scanned.

The selectivity of each predicate is estimated as one-tenth if the predicate is = else one-third. The total low-order selectivity is the product of the selectivity of each predicate. For example, the low-order selectivity of two = predicates is  $1/10 * 1/10 = 1/100$ .

The process of evaluating low-order predicates is called index scanning.

**DATA SELECTIVITY**

This is the filtering effect of predicates that must be evaluated from the data record. It is computed the same as low-order selectivity. This process is called data scanning. Since accessing data records is usually more expensive than index entries, data scanning is usually more expensive than index scanning of the same selectivity.

**INDEX**

This is the cost of reading the index. It is estimated as the number of index entries to be read divided by the average number of index entries per DXX block, plus the number of levels in the index.

$\text{cost} = \text{indexEntries} / \text{avg per blk} + \text{indexLevels}$

The number of index entries is computed as the number of entries indexed by high-order cardinality. For example, if high-order cardinality is 1,000 and there were 10000 rows indexed at the time the cardinality was computed, the number of index entries is  $10000 / 1000 = 10$ .

If OPTIMIZE FOR n ROWS is specified and it is less than the computed value, the number of index entries is reduced to this value.

### **DATA**

This is the cost of reading the data area. It is computed by dividing the number of rows to be read by the effective blocking factor of the index.

$$\text{cost} = \text{row} / \text{effBlk}$$

The number of data area rows is the number of index entries read times the selectivity of low-order predicates.

The effective blocking factor accounts for the randomness in which the data area is accessed by the index.

$$\text{effBlk} = \text{dataRows} * \text{BLKCHG} / 1024$$

where BLKCHG is the average number of times a new data block is encountered per 1024 rows, when accessed in the sequence of this index.

If at least 10 data blocks are to be read and the index is 90 percent in physical sequence, the cost is divided by two, to account for the savings of pre-fetch multi-block reads.

### **SORT**

This is the estimated cost of sorting for GROUP BY, ORDER BY, DISTINCT, or UNION. It applies only to the first join. If the first, or outer, table has an index that eliminates the need for a sort, its cost is not included.

### **ROWS**

This is the estimated number of rows in the result table. It is computed as the number of data area rows times data selectivity.

### **LOWEST ESTIMATED COST**

This is the estimated cost for the index with the lowest estimated cost.

Estimated cost is computed as the sum of the index, data and sort costs, multiplied by the number of rows from the previous join, or for the first join, the outer table restriction estimated rows.

*Merge*

KEY xxxxx MERGE CANDIDATE FOR TBL authId.tblName correlationName  
 KEY xxxxx SELECTED FROM MULTIPLE CANDIDATES

**MERGE CANDIDATE**

All equijoin columns are leading columns of the index.

**SELECTED CANDIDATE**

When there are several index merge candidate indexes, this message indicates the index selected. An index may be selected over another candidate for a lower total cost due to its eliminating sorts or a lower index and data scan cost.

SORT MERGE JOIN COSTS:  
 NO JOIN CONDITION - MERGE NOT POSSIBLE  
 GROUP/ORDER/DISTINCT/UNION SORT COST=nnnnnnnnnn  
 OUTER TABLE COSTS (SCAN,SORT,READ)=nnnnnnnnnn, nnnnnnnnnn, nnnnnnnnnn  
 INNER TABLE COSTS (SCAN,SORT,READ)=nnnnnnnnnn, nnnnnnnnnn, nnnnnnnnnn

**NO JOIN CONDITION**

No join condition was specified. Nested-loop method is selected.

**GROUP/ORDER/DISTINCT/UNION SORT COST**

Cost of sorting due to GROUP BY, and/or ORDER BY, UNION or DISTINCT.

**OUTER TABLE COSTS:****SCAN**

Restriction cost (cost of reading base table)

**SORT**

Cost of sorting (zero if index-merge)

**READ**

Cost of reading sorted temporary table (zero if index-merge)

**INNER TABLE COSTS**

Same as outer table.

TOTAL SORT-MERGE COST =nnnnnnnnnn  
 INDEXED MERGE COST USING INDEXES xxxxx AND xxxxx: nnnnnnnnnn

Only the message for the method with the lowest estimated cost is given.

**Join Step Summary**

MANUAL JOIN OPTIMIZATION SPECIFIED - NESTED LOOP USED  
 DISJUNCTIVE JOIN CONDITION - NESTED LOOP USED

**MANUAL JOIN OPTIMIZATION**

Plan option OPT=M is specified, so tables are joined in the order listed in the FROM clause and nested-loop method is used.

**DISJUNCTIVE JOIN CONDITION**

Join predicates are under an OR, so merge methods cannot be considered.

\*\*\* JOIN STEP LOWEST COST = nnnnnnnn

**JOIN STEP LOWEST COST**

The lowest estimate of all join methods and indexes.

## Execution-Time Messages

**Scope**

Execution-time messages report query execution statistics for the following statement types:

- OPEN and FETCH CURSOR
- INSERT (single row and searched)
- UPDATE (positioned and searched)
- DELETE (positioned and searched)

**Purpose**

You can use these execution statistics to determine:

- Which statements are using the most resources.
- Determine which query execution processes within a statement that is using the most resources.
- Compare bind-time resource estimations with actual performance.
- Compare resource usage between trial versions of a query.

## Operation

Execution-time messages are written to the SYSADM.SYSMSG table with sequence numbers higher than bind-time messages for a plan. A maximum of 32767 messages can be written for each statement of a plan.

Execution-time messages may be requested at either the summary or detail level:

- Summary-level messages are written when a plan is closed.
- Detail-level messages are written after the statement is executed or, for a cursor, when it is closed. For example, if you set plan option PLNCLOSE=T for a plan in a CICS transaction, these messages will be written for each CICS transaction or explicit COMMIT or ROLLBACK. To get a summary of all activity against the plan, use PLNCLOSE=R. This will cause the plan to only be closed when the SQL User Requirements Table is closed.

**WARNING!** Because of the number of rows inserted into the SYSADM.SYSMSG table by detail level or summary level CICS with PLNCLOSE=T, you may experience poor performance with these options.

## Execution-Time Summary Messages

This section describes the summary execution messages written to the SYSADM.SYSMSG table when a plan is closed.

There is a header message for changes in plan, and with plan for each statement, followed by detail messages for each table.

Messages are grouped by plan and statement, but they are ordered in the reverse sequence in which they were first executed.

### Plan Header

```
*** EXECUTION STATS FOR PLAN authId.planName
```

### Statement Header

```
STMT NBR: nnnnnnnnn TYPE: xxxxxxxxxxxx EXECUTIONS: nnnnnnnnn ROWS: nnnnnnnnn
```

**TYPE**

Type of statement as indicated by the following table:

**Type Code Process**

**FETCH CRS**

Open/Fetch Cursor

**SELECT INTO**

Select Into

**INSERT ROW**

Insert Row

**INSERT SET**

Insert Searched

**UPDATE CRS**

Update Positioned

**UPDATE SET**

Update Searched

**DELETE CRS**

Delete Positioned

**DELETE SET**

Delete Searched

**EXECUTIONS**

Number of times the statement was executed

**ROWS**

Number of rows fetched, inserted, updated, or deleted.

**Table Level**

These messages are in order by query block level and position, and table process sequence.

```
LVL NBR ----- TABLE NAME ----- --SETS-- --INDEX- --DATA-- --QUAL-- ---I/O--
nnn nnn xxxxxxxxxxxxxxxxxxxxxxxxxxxx nnnnnnnn nnnnnnnn nnnnnnnn nnnnnnnn nnnnnnnn
```

**LVL and NBR**

These fields indicate the query block in which the table is used. LVL is zero for subselects, and for subqueries indicates the level of nesting. When there are multiple select blocks at the same level, NBR indicates the order in which the select block appears in the SQL statement.

**TABLE NAME**

The base table name or kind of temporary table. If you have referenced the same table multiple times, or tables with the same name but different authId, in the same FROM clause, these messages are in the order the tables appear in the FROM clause. If you referenced a view with a join, the view reference has been expanded into a reference for each table in the view's FROM clause.

**SETS**

If the statement is executed multiple times, or the table is an inner table in a nested-loop join, or in a correlated subquery, this is the number of times a base table has been searched or a temporary table built. Divide SETS by statement EXECUTIONS to compute SETS per execution.

**INDEX**

The total number of index entries read.

**DATA**

The total number of data records read. The difference between INDEX and DATA is the number of rows rejected by low-order predicates, and the additional index intersection probes made if index intersection is used.

**QUAL**

The total number of qualified by Compound Boolean Selection (CBS) rows. Any rows rejected by predicates evaluated by the SQL subsystem, are not included. The difference between QUAL and DATA is the number of rows rejected by data scanning, that is to say, predicates evaluated by the Compound Boolean Selection Facility that reference a column that is not in the scan index. Also, if index merging is used, the difference includes the elimination of duplicates, where the same row was found in multiple indexes.

**ACCEPTED**

The number of rows that passed all the filtering criteria. Rows rejected by predicates evaluated by the SQL subsystem is computed by QUAL - ACCEPTED.

**I/O**

The total physical read and write I/O commands charged to requests accessing the table. A read-only task may be charged write I/Os if the buffer it needs is in a write pend status.

The ratio (INDEX + DATA) divided by I/O will tend to be the lowest for sequential access in data area physical sequence. Random access requires more I/O because each level of the index may need to be read for each row.

## Execution-Time Detail Messages

**Base Tables**

ACTIVITY FOR TBL <authId.tblName correlation name>

SETS=nnnnnnnnn INDEX=nnnnnnnnn DATA=nnnnnnnnn QUAL=nnnnnnnnn I/O=nnnnnnnnn

See the table level summary message for the definition of these fields.

CBS OPTIMIZER REASONS: xxxxxxxxxxxxxxxxxxxxxxxxx

Compound Boolean Selection (CBS) Optimizer Reasons indicate the type of processes used and why they were necessary.

For example, there is a query that you do not think should require index or data scanning, but INDEX and DATA is approximately twice QUAL. This can be explained if index merging is used because it builds a temporary index, where each data row may be accessed one or more times to build the temporary index, and then using the temporary index, read once more to return rows after duplicates have been eliminated. You can determine if index merging was indeed used from CBS Optimizer Reason number 4.

See the CBS Diagnostic Report description in the *CA Datacom/DB DBUTLTY Reference Guide*. for the definition of the optimizer reasons.

**Sort Temporary Tables**

SORT COMPLETED: ROWS EST=nnnnnnnnn IN=nnnnnnnnn, OUT=nnnnnnnnn

**EST**

Number rows estimated, which may be zero if an estimate was not required.

**IN**

Number rows input to the sort.



**OUT**

Number rows output by the sort. If GROUP BY or DISTINCT is specified, this can be a lower number than rows input to the sort because the sort process performs grouping and eliminates duplicate rows.

HIGHEST SORT MERGE LEVEL = nn

The sort first builds strings of sorted rows, then after several strings are built, the strings merged into a longer string. After several of these longer strings are built, they are in turn merged into yet a still longer string. This process is continued until there is a single result string. Each merging of strings is called a merge *level*. As you can see, the cost per row of a sort is dependent on the number of these merge levels.

**Quantified Subquery**

```
ACTIVITY FOR TEMP TBL **SUBQUERY nn **  
** SUBQUERY nnn ** TBLS=nnnnnnnnn ROWS=nnnnnnnnn READS=nnnnnnnnn
```

**SUBQUERY**

Number within SELECT block

**TBLS**

Number of times temporary table was built, that is to say, number of times subquery was executed. This will of course be 1 or 0 unless subquery is correlated.

**ROWS**

Number of rows in table. If TBLS is greater than 1, this is the sum of all the rows in all tables.

**READS**

Number of rows read from table. This can be greater than the number of rows if subquery is not correlated.

## Examples

### MSG=DD

In this example, the user is only concerned with customers with `IND_CD = 'A'`, so `IND_CD` was omitted from the `SELECT` and `GROUP BY` list. Since `CUST_NO` only appears as the second column of index `ACSTI`, this causes a sort for the `GROUP BY`. Adding `IND_CD` eliminates this inefficiency.

### Before Adding `IND_CD` to `SELECT` and `GROUP BY` List

```

PLAN:SYSADM          .ISDBXXX80270477   STM:000000001. DT: 07/28/2014 11.51.49
DECLARE C0          CURSOR FOR
SELECT CUST_NO, SUM(ORD_AMT)
FROM ACCTS
WHERE IND_CD = 'A'
GROUP BY CUST_NO
***** BIND MESSAGES FOR SUBSELECT 01 *****
INDEX DEFINITIONS FOR: ACT/010 SYSADM.ACCTS
KEYS= 2, IDXLVLS=1, LN= 31, ROWS/BLK= 95 ROWS= 3
  KEY ACTOR id= 4 FLG=D0 00 FLDS= 1 DXX= 300 BLKCHG= 204 ROWS=3
  OFFSET= 0, LN= 5, DIR=ASC SENS=N CARD= 3 ORD_ID
  KEY ACSTI id= 1 FLG=04 00 FLDS= 2 DXX= 300 BLKCHG= 204 ROWS=3
  OFFSET= 26, LN= 1, DIR=ASC SENS=N CARD= 2 IND_CD
  OFFSET= 27, LN= 4, DIR=ASC SENS=N CARD= 3 CUST_NO
TBL ACT DBID 010 RESTRICTION COSTS:
  KEY ACTOR 1ST 0 FLDS SELECTIVITY= 1.000000000
  0 LOW-ORDER FLDS SELECTIVITY= 1.000000000
  DATA SELECTIVITY= 0.099999964
  INDEX 1 DATA 1 SORT 3 ROWS 1
  KEY ACSTI 1ST 1 FLDS SELECTIVITY= 0.500000000
  0 LOW-ORDER FLDS SELECTIVITY= 1.000000000
  DATA SELECTIVITY= 1.000000000
  INDEX 1 DATA 1 SORT 3 ROWS 2
  *** KEY ACTOR HAS LOWEST ESTIMATED COST OF 5
  *** RESTRICTION COST = 5
ESTIMATED GROUP SORT COST= 3
KEY ACTOR HAS LOWEST COST OF 5 FOR 1 ROWS
GROUP BY SRT REASONS= CBS ORDER
BUILDING SORTED RESULT TABLE FOR SUBSELECT 1
ACTIVITY FOR TBL <SYSADM.ACCTS
SETS=000000001 INDEX=000000004 DATA=000000004 QUAL=000000004 I/O=000000000
CBS OPTIMIZER REASONS: < P P Y >
SORT COMPLETED: ROWS EST=1 IN=1, OUT=1
HIGHEST SORT MERGE LEVEL = 0

```

## After Adding IND\_CD to SELECT and GROUP BY List

```

PLAN:SYSADM          .ISDBXXX80270477  STM:000000002 DT: 07/28/2014 11.51.50
DECLARE C0          CURSOR FOR
SELECT IND_CD, CUST_NO, SUM(ORD_AMT)
FROM ACCTS
WHERE IND_CD = 'A'
GROUP BY IND_CD, CUST_NO
**** BIND MESSAGES FOR SUBSELECT 01 ****
INDEX DEFINITIONS FOR: ACT/010 SYSADM.ACCTS
KEYS= 2, IDXLVLS=1, LN= 31, ROWS/BLK= 95 ROWS= 3
KEY ACTOR id= 4 FLG=00 00 FLDS= 1 DXX= 300 BLKCHG= 204 ROWS=3
  OFFSET= 0, LN= 5, DIR=ASC SENS=N CARD= 3 ORD_ID
KEY ACSTI id= 1 FLG=04 00 FLDS= 2 DXX= 300 BLKCHG= 204 ROWS=3
  OFFSET= 26, LN= 1, DIR=ASC SENS=N CARD= 2 IND_CD
  OFFSET= 27, LN= 4, DIR=ASC SENS=N CARD= 3 CUST_NO
KEY ACSTI SATISFIES GROUP BY
TBL ACT DBID 010 RESTRICTION COSTS:
KEY ACTOR 1ST 0 FLDS SELECTIVITY= 1.000000000
  0 LOW-ORDER FLDS SELECTIVITY= 1.000000000
  DATA SELECTIVITY= 0.099999964
INDEX 1 DATA 1 SORT 3 ROWS 1
KEY ACSTI 1ST 1 FLDS SELECTIVITY= 0.500000000
  0 LOW-ORDER FLDS SELECTIVITY= 1.000000000
  DATA SELECTIVITY= 1.000000000
INDEX 1 DATA 1 SORT 0 ROWS 2
*** KEY ACSTI HAS LOWEST ESTIMATED COST OF 2
*** RESTRICTION COST = 2
ESTIMATED GROUP SORT COST= 3
KEY ACSTI USED FOR GROUPING
ACTIVITY FOR TBL <SYSADM.ACCTS >
SETS INDEX DATA QUAL ACCEPTED I/O CBS OPT REASONS
00001 000000004 000000004 000000004 00000000 0000000 < P P Y >

```



# Appendix B: Accessibility Features

---

The Schema Information Tables (SIT) are located in the SIT area, base 15, and are associated with an authorization ID of SYSADM, for example SYSADM.SYSCONSTRDEP, SYSADM.SYSCONSTROBJ, and so on. They contain CA Datacom/DB system information that can be queried by *authorized* users. **The SIT information should be properly secured.**

**Note:** For information about querying the SIT, see the *CA Datacom/DB Database and System Administration Guide*. For information about the SIT tables themselves, see the chapter about SQL tables (that also contains information about the SQL status tables) in the *CA Datacom/DB System Tables Reference Guide*.



# Appendix C: Sample Data Tables

---

The sample tables in this section are used in the examples in the section discussing application tasks that use embedded SQL.

## CUSTOMERS Table: Sample Data

		CUSTOMERS								
CUSNO	NAME	CITY	ST	ZIP	CRED OPEN \$	YTD SALES	ACT DT	SLMN ID		
0030	CANNON TOOLS CO	ATLANTA	GA	303012334	A	23442.00	3322123.00	860220 34222		
0090	INTERNATIONAL BANK CORP	NEW YORK	NY	100059989	A	.00	211650.00	850815 21165		
0130	SUN DIAL CITRUS GROWERS	LOS ANGELES	CA	902130000	A	21489.00	293900.00	850815 29390		
0150	IMPERIAL BANKCORP	NEW YORK	NY	100190000	A	.00	131500.00	850815 13150		
0170	UNITED ATLANTIC SHARES	CHARLOTTE	NC	282552550	A	.00	314200.00	850815 31420		
0190	SOUTHWEST STATE OIL REFINING	SHERMAN OAKS	CA	914231423	A	.00	179800.00	850815 17980		
0210	WEST LIFE INSURANCE	BALTIMORE	MD	212031203	A	.00	6550.00	850815 00655		
0230	CHEMICAL MUTUAL	FORT WORTH	TX	761026102	A	931.72	7250.00	860220 00725		
0250	MICHIGAN LIGHTING INC.	PHOENIX	AZ	850365036	A	.00	182600.00	850815 18260		
0270	HARTFORD IRON WORKS INC	ALLENTOWN	PA	181018101	A	.00	7600.00	850815 00760		
0290	NATIONAL HARRIS CORPORATION	ATLANTA	GA	303310331	A	.00	7950.00	850815 00795		
0310	CARTERET STEEL CORPORATION	SAN MATEO	CA	940024002	A	3457.00	10750.00	850815 01075		
0330	HARVESTLAND FOODS INC	HONOLULU	HI	968426842	A	.00	186100.00	850815 18610		
0350	FLEET BANKING INDUSTRIES	PORTLAND	OR	972047204	A	.00	11450.00	850815 01145		
0370	GREAT LAKES PRODUCTS	PLAINFIELD	IN	461686168	A	.00	186450.00	850815 18645		
0390	HOLLY TEXTILES INTERNATIONAL	WORCESTER	MA	016041604	A	.00	187150.00	850815 18715		
0410	COASTLINE INC	BOSTON	MA	021992199	A	528.84	187850.00	850815 18785		
0430	WESTER CORPORATION	ST. LOUIS PARK	MN	554265426	A	.00	190300.00	850815 19030		
0450	HORIZON LABORATORIES	MILWAUKEE	WI	532023202	A	.00	15650.00	850815 01565		
0470	FIRST DOWNEY INC	NORFOLK	VA	235103510	A	.00	16000.00	850815 01600		
0490	MORRISON HOME FEDERATION	PROSPECT HGTS	IL	600740074	A	.00	194850.00	850815 19485		
0510	SIGNAL/WEST PETROLEUM	LOS ANGELES	CA	900540054	A	.00	204300.00	850815 20430		
0530	GULF LAND USA	DALLAS	TX	752345234	A	.00	207100.00	850815 20710		
0550	H. F. MURPHY CORP	EL DORADO	AR	717301730	A	.00	34550.00	850815 03455		
0570	TEXAS LIFE & CASUALTY CO	DALLAS	TX	752225222	A	.00	34900.00	850815 03490		
0610	SUN FIBERGLASS	MIAMI BEACH	FL	331413141	A	.00	40150.00	850815 04015		
0630	BAY-BANK AUTOMOBILES	NASHVILLE	TN	372027202	A	530.50	41550.00	850815 04155		
0650	CONTINENTAL GREETINGS INC	TWINSBURG	OH	440874087	A	.00	45750.00	850815 04575		
0670	TRANSAMERICA RUBBER INDUSTRIES	CLEARWATER	FL	335183518	A	.00	46100.00	850815 04610		
0690	MANUFACTURERS LABORATORIES	OAKBROOK	IL	605210521	A	.00	46450.00	850815 04645		
0710	NORTHERN MEDICAL SERVICES	ROCHELLE PARK	NJ	076627662	A	.00	47150.00	850815 04715		
0730	TRIBUNE MOTORS	SALT LAKE CITY	UT	841274127	A	.00	47500.00	850815 04750		
0750	CAMERON INCORPORATED	WILMINGTON	DE	198019801	A	.00	49250.00	850815 04925		
0770	PROVIDENCE CHEMICAL COMPANY	NEW YORK	NY	100150015	A	.00	49600.00	850815 04960		
0790	REYNOLDS SHIPPING	WICHITA	KS	672187218	A	.00	229850.00	850815 22985		
0810	SMITHFIELD TRANSPORTATION	NEW YORK	NY	100220022	A	.00	230200.00	850815 23020		
0830	AVERY PRINTING	SPRINGDALE	AR	727642764	A	.00	230900.00	850815 23090		
0850	NUSON DRUGS & RESEARCH	OKLAHOMA CITY	OK	731253125	A	.00	56250.00	850815 05625		
0870	UNIVERSAL AIRWORKS	ARMONK	NY	105040504	A	.00	231600.00	850815 23160		
0890	BELL-BAKER POWER & SERVICE	BOISE	ID	837263726	A	.00	57300.00	850815 05730		
0910	C.F. RIVERS AIRLINE & FREIGHT	PITTSBURG	PA	152305230	A	.00	58700.00	850815 05870		
0930	FLEETWOOD FREIGHTWAYS INC	WILMINGTON	DE	198019801	A	.00	235450.00	850815 23545		
0950	M.A.C. SAVINGS	ATLANTA	GA	303020302	A	.00	236150.00	850815 23615		
0970	PHELPS FINANCIAL CORP	VAN NUYS	CA	914051405	A	1441.08	239650.00	850815 23965		
0990	SUNSTRAND BANKS	MEMPHIS	TN	381038103	A	.00	241400.00	850815 24140		
1010	INTERNATIONAL SHIPYARDS INC	CHARLOTTE	NC	282558255	A	.00	242450.00	850815 24245		
1030	BROTHERS GYPSUM INC	MIAMI	FL	328012801	A	.00	69550.00	860220 06955		
1050	MACMILLAN HOME PRODUCTS CO	COSTA MESA	CA	282888288	A	.00	248400.00	850815 24840		
1070	JORIE PAPER INC	PHOENIX	AZ	850125012	A	.00	76200.00	850815 07620		
1090	CHESTERSON-KIDD INC	BALTIMORE	MD	212021202	A	.00	251550.00	850815 25155		
1110	LEXINGTON CHEMICAL	ST PETERSBURG	FL	972047204	A	.00	254350.00	850815 25435		
1130	ST. LOUIS FOODS DIST.	DENVER	CO	100190019	A	.00	258550.00	850815 25855		
1150	AFTON INDUSTRIES	DALLAS	TX	752405240	A	.00	259250.00	850815 25925		
1170	PARK-HESS & COMPANY	GLENDALE	CA	912031203	A	.00	263100.00	850815 26310		
1190	KELLWOOD-HANDY TRUCKING	MOBILE	AL	366096609	A	.00	263800.00	850815 26380		



1210	LINGBERGH INDUSTRIES	SEATTLE	WA	981858185	A	2032.10	89150.00	860220	08915
1230	BORDEN-SPEERY INSTRUMENTS	NEW ORLEANS	LA	701600160	A	.00	264150.00	850815	26415
1250	PALMOLIVE INNS	DALLAS	TX	752015201	A	.00	89500.00	850815	08950
1270	SOUTHLAND DEPARTMENT STORES	FORT WORTH	TX	761016101	A	.00	90900.00	850815	09090
1290	MCDONNEL SYSTEMS	WOODLANDS	TX	773807380	A	.00	91250.00	850815	09125
1310	RESOUC COMPUTERS	OIL CITY	PA	163016301	A	.00	272550.00	850815	27255
1330	NETTLETON AIRCRAFT	BARTLESVILLE	OK	740044004	A	.00	97550.00	850815	09755
1350	HARMAN-MCGEE INDUSTRIES	WHITE PLAINS	NY	106500650	A	.00	99650.00	850815	09965
1370	MCCORMICK ELECTRIC SUPPLIES	HOUSTON	TX	772517251	A	.00	275000.00	850815	27500
1390	MIDLAND CHEMICAL	HAMILTON	OH	452025202	A	.00	275700.00	850815	27570
1410	WRIGHT-ILLINOIS DATA	FRESNO	CA	937213721	A	.00	276750.00	850815	27675
1430	PITWM-WAY FOODS	WASHINGTON	DC	200760076	A	.00	277100.00	850815	27710
1450	UNION TRANSPORTATION	GALVESTON	TX	775507550	A	1807.30	277450.00	860220	27745
1470	MARK & MARK INTERNATIONAL	BETHPAGE	NY	117141714	A	.00	104200.00	850815	10420
1490	SALEM PAPERBOARD	YORKLYN	DE	197369736	A	.00	280600.00	850815	28060
1510	ABBOTT BRANDS CORP	HUNT VALLEY	CA	900360036	A	.00	105600.00	850815	10560
1530	JOHNSON MULTIFOODS	BETHESDA	VA	232613261	A	.00	105950.00	850815	10595
1550	DELUX COSMETICS	HAWTHORNE	SC	295509550	A	.00	283400.00	850815	28340
1570	PAMMEL BREWING	DALLAS	TX	752015201	A	.00	109800.00	850815	10980
1590	FOXBORRO PETRO-CHEMICAL	GERMANTOWN	MD	212011201	A	.00	286550.00	850815	28655
1610	ROLM-CASTLE SOY PRODUCTS	SANTA MONICA	CA	904060406	A	.00	112600.00	850815	11260
1630	MARBURY MATERIALS	BURBANK	CA	911031103	A	446.50	287600.00	860220	28760
1650	METROPLEX ASSOCIATED	RICHMOND	VA	232203220	A	.00	112950.00	850815	11295
1670	MALIRY ENTERTAINMENT INDUSTRY	BALTIMORE	MD	915201520	A	.00	114000.00	850815	11400
1690	WALTERS-BORUM INCOP	PASADENA	CA	911031103	A	.00	289350.00	850815	28935
1710	PARKER REPUBLIC CONSOLIDATED	TOWSON	MD	201240124	A	.00	117850.00	850815	11785
1730	HOOVER-EAGLE TIRE & RUBBER	GREENSBORO	NC	274207420	A	.00	293550.00	850815	29355
1750	MILES-COOPER GENERAL CORP.	CHULA VISTA	CA	232203220	A	.00	118900.00	850815	11890
1770	MULTMOMAH CORPORATION	WINSTON-SALEM	NC	271027102	A	.00	121000.00	850815	12100
1790	DORSEY STEEL INC	HARTSVILLE	SC	295509550	A	.00	121350.00	850815	12135
1810	EATON-AKRON PACIFIC CORP	VALLEY FORGE	PA	194829482	A	.00	297050.00	850815	29705
1830	SPRINGS-WEYER LABORATORIES	RIVERSIDE	CA	911031103	A	.00	297750.00	850815	29775
1850	TECH CASTLE RESEARCH	GREENSBORO	NC	274207420	A	.00	298800.00	850815	29880
1870	WULCAR-FORMAN REAL ESTATE INMT	FORT WORTH	TX	720912091	A	.00	299150.00	850815	29915
1890	FIRST STREET BANK CORP	COLUMBUS	OH	430853085	A	2717.00	124850.00	860220	12485
1910	GRAND RAPIDS FINANCIAL ASSOC	LOUISVILLE	KY	402230223	A	.00	125550.00	850815	12555
1930	NORTHERN EXPRESS	INDIANAPOLIS	IN	462856285	A	.00	125900.00	850815	12590
1950	WESTMINISTER MEDICAL SUPPLIES	TOLEDO	OH	436973697	A	.00	301600.00	850815	30160
1970	KENT SAVINGS & LOAN	LAKEWOOD	OH	441074107	A	.00	302650.00	850815	30265
1990	MOSSVINE FOODS	WICKLIFFE	OH	440924092	A	.00	129050.00	850815	12905
2010	HOLIDAY MERCHANDISE INC.	CANTON	OH	447114711	A	.00	304400.00	850815	30440
2030	KNIGHT-GANNETT TOYS INC	HARTFORD	CT	494439443	A	.00	129400.00	850815	12940
2050	TRANSAMERICAN PUBLISHING	TELMUSEH	MI	492769276	A	5127.93	304750.00	860220	30475
2070	NICOLLET STORES INC.	MILWAUKEE	WI	532013201	A	.00	305100.00	850815	30510
2090	BRYAN DAIRY PRODUCTS	ORANGE	CT	064776477	A	.00	305800.00	850815	30580
2110	SERVICE AGRICULTURE INC	ST. PAUL	MN	551085108	A	.00	132200.00	850815	13220
2130	INTERSALE INC.	GLENDALE	WI	532013201	A	.00	136050.00	850815	13605
2150	WESTLAND DEPT STORES	NEWTON	IA	502080208	A	.00	138500.00	850815	13850
2170	HERALD DRY GOODS	TECUMSETT	MI	492769276	A	.00	139550.00	850815	13955
2190	OSWISH MANUFACTURING	WOODCLIFF LAKE	NJ	076757675	A	.00	139900.00	850815	13990
2210	ALLEGHENY HOTELS	CHICAGO	IL	606060606	A	.00	141650.00	850815	14165
2230	AMTEX FREIGHTWAYS	NEW YORK	NY	100160016	A	.00	142350.00	850815	14235
2250	ALLIED CONTINENTAL	BRIDGEWATER	NJ	079607960	A	.00	317700.00	850815	31770
2270	HANKS-MAY INVESTMENTS INC	SKOKIE	IL	600770077	A	.00	143400.00	850815	14340
2290	XSMART CINEMAS	KANSAS CITY	MO	631783178	A	.00	144100.00	850815	14410
2310	UNIONWAYS TRANSPORT CO.	PEORIA	IL	616291629	A	.00	144450.00	850815	14445
2330	CORTER-SOUTH ACCOUNTING	ST. LOUIS	MO	631783178	A	.00	320850.00	850815	32085
2350	ASSOCIATED DIXIE RESOURCES	MOLINE	IL	616291629	A	.00	146550.00	850815	14655
2370	LANDMARK FOODS	DECATOR	IL	625252525	A	.00	148650.00	850815	14865

## ORDERS Table: Sample Data

ORDID	CUSTNO	ORD DT	EXP DT	ORDERS		SHP DT	TERMS	SHIP	ORD TOT	FRT AMT	SLMN	LAST DT
				DSC	STATUS							
01008	7290	851106	85121	2.0	S	860220	NET 30	WWWWW	360.50	2.99	13185	860220
01009	7350	851106	851213	3.0	S	860220	NET 30		979.93	2.50	13745	860220
01010	2690	851106	85121	2.0	S	860220	NET 30		8073.05	8.00	15740	860220
01011	0030	851106	851215	3.0	P		NET 30	4250.30	.00	17630	860220	
01012	7410	851108	851215	.0	P		NET30	305.84	.00	21060	860220	
01013	0230	851108	851210	.0	P		NET30	931.72	.00	00725	860220	
01015	1210	851108	851214	.0	P		NET30	2032.10	.00	08915	860220	
01016	1450	851108	851201	.0	A		NET30	1807.30	.00	27745	860220	
01017	1630	851108	851230	.0	A		NET30	446.50	.00	28760	860220	
01018	1890	851108	851218	.0	A		NET30	2717.00	.00	12485	860220	
01019	2050	851108	851208	.0	A		NET30	4641.93	.00	30475	860220	
01020	7790	851108	851202	.0	A		NET30	2710.61	.00	04085	860220	
01021	3910	851108	851210	.0			NET30	3595.75	.00	07165	860220	
01022	5590	851108	851213	.0			NET30	2258.10	.00	07970	860220	
01023	7150	851108	851212	.0			NET30	190.65	.00	08600	860220	
01024	4310	851108	851225	.0			NET30	76.00	.00	06080	860220	
01026	9130	851108	851203	.0			NET30	181.78	.00	06640	860220	
01027	4350	851108	851214	.0			NET30	146.90	.00	19135	860220	
01028	1210	851108	851230	.0			NET30	1645.65	.00	28830	860220	
01029	1850	851108	851220	.0			NET30	745.74	.00	13255	860220	
01030	6390	851108	851216	.0			NET30	353.50	.00	01845	860220	

# Appendix D: Results of Defining Structures Using SQL Statements

---

The following describe how using SQL statements to define structures impacts the CA Datacom Datadictionary definition. For each SQL syntax segment, only the significant attribute-values are discussed. See the Occurrence Names Created from SQL Names section in the *CA Datacom Datadictionary User Guide* for details on actions taken by CA Datacom Datadictionary to create unique names and the *CA Datacom Datadictionary Attribute Reference Guide* for details on attribute values of the various entity-types.

## CREATE INDEX Statement

### **CREATE INDEX index-name (column-list)**

Creates a key for one or more columns of a table as specified in the statement. The following are the significant KEY entity-occurrence attributes defined:

- AGR-SQLNAME= The SQL name of the table.
- AUTHID= The schema (authorization identifier) of the table.
- DATACOM-ID= An available 3-digit ID selected by Datadictionary for uniqueness within the database.
- DATACOM-NAME= The 5-character value specified with the optional DATACOM NAME parameter in the CREATE INDEX statement or SQ followed by three digits selected by Datadictionary for uniqueness within the database.
- ENTITY-NAME= The *index-name* specified in the CREATE INDEX statement.
- FOREIGN=N
- INCLUDE-NIL-KEY=Y
- MASTER-KEY=N
- MAX-KEY-LENGTH=0
- NATIVE-KEY=N
- SQLNAME= The *index-name* provided on the CREATE INDEX statement.
- UNIQUE=N

## CREATE PROCEDURE Statement

### CREATE PROCEDURE

Creates a procedure. The following are the significant PROCEDURE entity-occurrence attributes defined:

- AUTHID= The authorization identifier (the schema) in effect when the SQL CREATE statement was executed.
- CREATOR= The accessor ID assigned at the SQL CREATE statement execution.
- DETERMINISTIC= Indicates that the procedure does (Y) or does not (N) return the same value given the same input (whether a procedure can provide different results due to factors other than the data). This attribute is informational only.
- ENTITY-NAME= This is an internally generated unique name consisting of the schema (the AUTHID), followed by a hyphen, followed by the SQLNAME value.
- EXTERNAL= If the procedure causes an external module to be executed, this attribute is set to Y. Otherwise, if the procedure directly executes SQL statements, this attribute is set to N.
- FIPS= Indicates if the procedure is valid in FIPS processing. Y indicates yes, N indicates no, and S indicates no because of syntax.
- LANGUAGE= The programming language used to produce the module executed when the procedure is invoked.
- PARM-LIST-STYLE= The style of the parameter list to be passed to the procedure module by SQL: GENERAL, GENERAL WITH NULLS, or DATACOM SQL.
- PRC-VALID=Y When a change is made to a program referenced by a procedure, this attribute is set to N.
- SQLNAME= The name of the procedure as specified in the CREATE PROCEDURE statement.
- SQL-ACCESS= Indicates whether the procedure executed contains SQL statements and whether SQL performs only read function (READ), read and updated functions (MODIFY), or only other SQL functions such as DDL (YES). If no SQL is executed, the value is NO.

## CREATE SCHEMA Statement

### **CREATE SCHEMA AUTHORIZATION *authid***

Creates a schema, also known as an AUTHID, which is defined in CA Datacom Datadictionary as an AUTHORIZATION entity-occurrence. The following are the significant AUTHORIZATION entity-occurrence attributes defined:

- AUTH-USAGE=S
- ENTITY-NAME= The name (*authid*) specified in the CREATE SCHEMA statement.

## CREATE TABLE Statement

### **CREATE TABLE *table-name***

Creates a table, a key, and an element for the entire table. It relates the TABLE entity-occurrence to the AUTHORIZATION entity-occurrence (the schema).

The following are the significant TABLE entity-occurrence attributes defined:

- AUTHID= The schema (authorization identifier) specified with a qualified *table-name* in the statement or the default schema.
- CHNG-MASTER-KEY=Y
- COMPRESSION=N
- CONSTRAINT=Y if the table or any of the columns has a constraint defined; otherwise, CONSTRAINT=N.
- DATACOM-ID= A 3-digit ID generated by CA Datacom Datadictionary that is unique within the database.
- DATACOM-NAME= An internally generated unique name in the form *Bnn*, *Cnn*, or *Dnn* where *nn* is 00 to 99, or the first three characters of the SQLNAME, if that is unique.
- DDD-SYNCH=Y
- DUPE-MASTER-KEY=Y
- ENTITY-NAME= An internally generated unique name consisting of the schema (the AUTHID used when creating the table) followed by a hyphen and the name specified in the CREATE statement. The name must be unique. See the Occurrence Names Created from SQL Names section in the *CA Datacom Datadictionary User Guide* for details on actions taken by CA Datacom Datadictionary to create unique names.

- FIPS= Indicates if the procedure is valid in FIPS processing. Y indicates yes, N indicates no, and S indicates no because of syntax.
- LOGGING=Y
- PIPELINE-OPTION=Y
- RECOVERY=Y
- SQL-INTENT=Y
- SQLNAME= The *table-name* provided in the CREATE TABLE statement.

The element contains all the columns defined to the table. The following are the significant ELEMENT entity-occurrence attributes defined:

- DATACOM-NAME=SQLEL
- DISP-IN-TABLE=0
- ENTITY-NAME=SQLEL
- FIRST-FIELD= The occurrence-name of the first field in the table.
- LAST-FIELD= The occurrence-name of the last field in the table.
- LENGTH= The length of the table including null indicators for columns defined to accept null values.

The first column named in the CREATE TABLE statement becomes the CA Datacom/DB Master and Native Key for the table except when a primary key is defined for the table. The following are the significant KEY entity-occurrence attributes defined:

- AGR-SQLNAME= The SQL name of the table.
- AUTHID= The schema (authorization ID) of the table.
- DATACOM-NAME= SQ followed by three digits selected by CA Datacom Datadictionary for uniqueness within the database (same name as used for ENTITY-NAME).
- DATACOM-ID= An available 3-digit ID selected by CA Datacom Datadictionary for uniqueness within the database.
- ENTITY-NAME= SQ followed by three digits selected by CA Datacom Datadictionary for uniqueness within the database (same name as used for DATACOM-NAME).
- FOREIGN=N
- INCLUDE-NIL-KEY=Y
- MASTER-KEY=Y unless a primary key is defined, in which case N.
- MAX-KEY-LENGTH=0
- NATIVE-KEY=Y unless a primary key is defined, in which case N.

- SQLNAME= The key's DATACOM-NAME followed by an underscore character followed by the concatenation of the DATACOM-NAME of the table and the DATACOM-ID of the database in which the key is defined (for example, SQ032\_INV00016).
- UNIQUE=Y unless a primary key is defined, in which case N.

**IN area-name**

Relates the TABLE entity-occurrence created to the AREA entity-occurrence named. If you do not include this parameter, the table is placed in the SQL default area (see the *CA Datacom/DB Database and System Administration Guide*).

**CREATE TABLE table-name column-name**

Defines a column (FIELD entity-occurrence) in the table. The following are the significant FIELD entity-occurrence attributes defined:

- AFTER=START for first column; otherwise, the ENTITY-NAME of the column it follows.
- AGR-SQLNAME= The SQL name of the table.
- AUTHID= The schema (authorization ID) of the table.
- CLASS=S
- ENTITY-NAME = The *column-name* specified in the CREATE TABLE statement (the same name as used for SQLNAME).
- LENGTH= The length of the column.
- PARENT=START
- SQLNAME= The *column-name* specified in the CREATE TABLE statement (the same name as used for ENTITY-NAME).

Additional attributes are defined based on the column definition syntax as follows.

**datatype**

Defines the column's data type through FIELD entity-occurrence attributes. If the column's data type is DATE, TIME, or TIMESTAMP:

- SEMANTIC-TYPE= This attribute is defined as SQL-DATE, SQL-TIME, or SQL-STMP, respectively.
- SIGN=N
- TYPE=B
  - length 4 for SQL-DATE
  - length 3 for SQL-TIME
  - length 10 for SQL-STMP
- TYPE-NUMERIC=C

**FOR BIT DATA**

For the FIELD entity-occurrence, specifies SEMANTIC-TYPE=BITDATA (only valid for character, VARCHAR, and LONG VARCHAR data types).

**DEFAULT literal**

For the FIELD entity-occurrence, defines a literal value in the VALUE attribute and sets the DEFAULT-INSERT attribute to O.

**DEFAULT USER**

For the FIELD entity-occurrence, sets the DEFAULT-INSERT attribute to U.

**DEFAULT NULL**

For the FIELD entity-occurrence, sets the NULL-INDICATOR attribute to Y and the DEFAULT-INSERT attribute to N.

**DEFAULT SYSTEM USER**

For the FIELD entity-occurrence, sets the DEFAULT-INSERT attribute to S.

**NOT NULL**

For the FIELD entity-occurrence, sets the NULL-INDICATOR attribute to N.



**NOT NULL WITH DEFAULT**

For the FIELD entity-occurrence, sets the DEFAULT-INSERT attribute to D.

**WITH DEFAULT**

For the FIELD entity-occurrence, sets the DEFAULT-INSERT attribute to D.

**PRIMARY KEY**

Creates a key consisting of this column or columns and a constraint. The following are the significant KEY entity-occurrence attributes defined:

- AGR-SQLNAME= The SQL name of the table.
- AUTHID= The schema (authorization ID) of the table.
- DATACOM-ID= An available 3-digit ID selected by CA Datacom Datadictionary for uniqueness within the database.
- DATACOM-NAME= SQ followed by three digits selected by CA Datacom Datadictionary for uniqueness within the database (same name as used for the ENTITY-NAME).
- ENTITY-NAME= SQ followed by three digits selected by CA Datacom Datadictionary for uniqueness within the database (same name as used for the DATACOM-NAME).
- FOREIGN=N
- INCLUDE-NIL-KEY=Y
- MASTER-KEY=Y
- MAX-KEY-LENGTH=0
- NATIVE-KEY=Y
- SQLNAME= The key's DATACOM-NAME followed by an underscore character followed by the concatenation of the DATACOM-NAME of the table and the DATACOM-ID of the database in which the key is defined (for example, SQ032\_INV00016).
- UNIQUE=Y

The following are the significant CONSTRAINT entity-occurrence attributes defined:

- AUTHID= The schema (authorization ID) associated with the table.
- ENTITY-NAME= An internally generated unique name consisting of the schema (the AUTHID) used when creating the table followed by a hyphen and the SQLNAME.
- SQLNAME= The name given to the constraint in the statement or an internally created name in the form CONSTRAINT\_ *nnnn* where the *nnnn* are four digits selected for uniqueness within the database.

The TABLE entity-occurrence has the following attributes that pertain to constraints:

- CNS-UNIQUE=Y
- CONSTRAINT=Y

### UNIQUE

Defines a key consisting of the identified column or columns and a constraint. The following are the significant KEY entity-occurrence attributes defined:

- AGR-SQLNAME= The SQL name of the table.
- AUTHID= The schema (authorization ID) of the table.
- DATACOM-ID= An available 3-digit ID selected by CA Datacom Datadictionary for uniqueness within the database.
- DATACOM-NAME= SQ followed by three digits for uniqueness within the database (same name as used for the ENTITY-NAME).
- ENTITY-NAME= SQ followed by three digits for uniqueness within the database (same name as used for the DATACOM-NAME).
- FOREIGN=N
- INCLUDE-NIL-KEY=Y
- MASTER-KEY=N if this is not the first unique key of the table and a primary key is not defined; otherwise, MASTER-KEY=Y.
- MAX-KEY-LENGTH=0
- NATIVE-KEY=N if this is not the first unique key of the table and a primary key is not defined; otherwise, NATIVE-KEY=Y.
- SQLNAME= The key's DATACOM-NAME followed by an underscore character followed by the concatenation of the DATACOM-NAME of the table and the DATACOM-ID of the database in which the key is defined (for example, SQ032\_INV00016).
- UNIQUE=Y

The following are the significant CONSTRAINT entity-occurrence attributes defined:

- AUTHID= The schema (authorization ID) associated with the table.
- ENTITY-NAME= An internally generated unique name consisting of the schema (the AUTHID used when creating the table) followed by a hyphen and the SQLNAME.
- SQLNAME= The name given to the constraint in the statement or an internally created name in the form CONSTRAINT\_ *nnnn* where the *nnnn* are four digits selected for uniqueness within the database.

The TABLE entity-occurrence has the following attributes that pertain to constraints:

- CNS-UNIQUE=Y
- CONSTRAINT=Y

#### **REFERENCES table-name (column-list)**

Sets the TABLE entity-occurrence CNS-REFERS= and CONSTRAINT= attributes to Y for this table.

Sets the referenced TABLE entity-occurrence CNS-REFERENCED= and CONSTRAINT= attributes to Y.

Also establishes a foreign key occurrence (see following page).

#### **CHECK (search condition)**

Creates a constraint. The following are the significant CONSTRAINT entity-occurrence attributes defined:

- AUTHID= The schema (authorization ID) associated with the table.
- ENTITY-NAME= An internally generated unique name consisting of the schema (the AUTHID used when creating the table) followed by a hyphen and the SQLNAME.
- SQLNAME= The name given to the constraint in the statement or an internally created name in the form CONSTRAINT\_ *nnnn* where the *nnnn* are four digits selected for uniqueness within the database.

The TABLE entity-occurrence has the following attributes that pertain to constraints:

- CNS-DOMAIN=Y
- CONSTRAINT=Y

#### **CONSTRAINT constraint-name**

Sets the CONSTRAINT entity-occurrence SQLNAME attribute to the name specified in the statement.

#### **FOREIGN KEY (column-list)**

Creates a key consisting of the identified column or columns and a constraint. The following are the significant KEY entity-occurrence attributes defined:

- AGR-SQLNAME= The SQL name of the table.
- AUTHID= The schema (authorization ID) of the table.
- AGR-SQLNAME= The SQL name of the table.

- AUTHID= The schema (authorization ID) of the table.
- DATACOM-NAME= blanks
- DATACOM-ID=000
- ENTITY-NAME= CA Datacom Datadictionary uses the constraint's ENTITY-NAME value.
- FOREIGN=Y
- INCLUDE-NIL-KEY=Y
- MASTER-KEY=N
- NATIVE-KEY=N
- MAX-KEY-LENGTH=00
- SQLNAME= CA Datacom Datadictionary uses the constraint's SQLNAME value.
- UNIQUE=N

The following are the significant CONSTRAINT entity-occurrence attributes defined:

- AUTHID= The schema (authorization ID) associated with the table.
- ENTITY-NAME= An internally generated unique name consisting of the schema (the AUTHID used when creating the table) followed by a hyphen and the SQLNAME.
- SQLNAME= The name given to the constraint in the statement or an internally created name in the form CONSTRAINT\_ *nnnn* where the *nnnn* are four digits selected for uniqueness within the database.

The TABLE entity-occurrence has the following attributes:

- CNS-SAME-BASE=Y (if all tables to which the foreign key refers are in the same base).
- CNS-REFERS=Y
- CONSTRAINT=Y

Sets the referenced TABLE entity-occurrence CNS-REFERENCED= and CONSTRAINT= attributes to Y.

## CREATE SYNONYM Statement

### **CREATE SYNONYM synonym-name FOR table-name (or view-name)**

Creates a synonym. The following are the significant SYNONYM entity-occurrence attributes defined:

- AUTHID= The supplied or default schema (authorization ID).
- ENTITY-NAME= An internally generated unique name consisting of the schema (the AUTHID) followed by a hyphen and the name specified in the CREATE statement (the name must be unique— see the Occurrence Names Created from SQL Names section in the *CA Datacom Datadictionary User Guide* for details on actions taken by CA Datacom Datadictionary to create unique names).
- SQLNAME= The *synonym-name* provided in the CREATE SYNONYM statement.

CA Datacom Datadictionary creates relationships to associate the table or view with its synonyms.

## CREATE TRIGGER Statement

### **CREATE TRIGGER**

Creates a trigger. The following are the significant TRIGGER entity-occurrence attributes defined:

- ACTION-LEVEL= In conjunction with the Trigger Action Time switches, indicates whether the trigger is to be invoked for each row (R) processed or for each SQL statement (S).
- AFTER-DELETE=  
AFTER-INSERT=  
AFTER-UPDATE=

Indicates with Y or N whether the trigger should be invoked after the action has taken place on the table. One or more of these switches and/or the BEFORE-x switches must be set to Y for the trigger to be valid.

- AUTHID= This is the authorization identifier (schema) in effect when the trigger was created.
- BEFORE-DELETE=  
BEFORE-INSERT=  
BEFORE-UPDATE=  

Indicates with Y or N whether the trigger should be invoked before the action has taken place on the table. One or more of these switches and/or the AFTER-x switches must be set to Y for the trigger to be valid.
- ENTITY-NAME= This is an internally generated unique name consisting of the schema (the AUTHID), followed by a hyphen, followed by the SQLNAME value.
- FIPS= Indicates if the procedure is valid in FIPS processing. Y indicates yes, N indicates no, and S indicates no because of syntax.
- NEW-ROW-CORR-NAME=  
OLD-ROW-CORR-NAME=  
The alternative names for the before and after row images.
- NEW-TABLE-ALIAS=  
OLD-TABLE-ALIAS=  
The alternative names for the before and after table images.
- SQLNAME= The name of the trigger specified in the CREATE TRIGGER statement.
- SQL-TIMESTAMP= The ANSI Standard specifies that triggers are to be invoked in the order they are created, with the oldest being invoked first. The SQL timestamp ensures that triggers are invoked in that order.
- TRIGGER-VALID= When a change is made to a table referenced by a trigger and that change can affect the ability of the trigger to function correctly, this attribute is set to N. Otherwise, it is set to Y. This attribute is also set to N if there is a recognized change to a procedure called by the trigger.
- UPDATE-TRIGGER= This attribute indicates, in conjunction with the BEFORE-UPDATE and AFTER-UPDATE attributes, whether the trigger is invoked only when certain columns are updated (EXPLICIT) or when any column in the table is updated (IMPLICIT).

---

## CREATE VIEW Statement

### **CREATE VIEW view-name**

Creates a view. The following are the significant VIEW entity-occurrence attributes defined:

- AUTHID= The supplied or default schema (authorization ID).
- ENTITY-NAME= An internally generated unique name consisting of the schema (the AUTHID) followed by a hyphen and the name specified in the CREATE statement (the name must be unique— see the Occurrence Names Created from SQL Names section in the *CA Datacom Datadictionary User Guide* for details on actions taken by CA Datacom Datadictionary to create unique names).
- FIPS= Indicates if the procedure is valid in FIPS processing. Y indicates yes, N indicates no, and S indicates no because of syntax.
- LENGTH= The sum of the lengths of all columns included in the view.
- SQLNAME= The *view-name* specified in the CREATE VIEW statement.

### **column-name**

Creates a column (FIELD entity-occurrence). See the CREATE TABLE *column-name* statement.

### **AS subselect**

Creates a relationship between the view and the table whose columns make up the view.

### **WITH CHECK OPTION**

Creates a domain constraint. The following are the significant CONSTRAINT entity-occurrence attributes defined:

- AUTHID= The schema (authorization ID) associated with the table.
- ENTITY-NAME= An internally generated unique name consisting of the schema (the AUTHID used when creating the table) followed by a hyphen and the SQLNAME.
- SQLNAME= The *view-name* provided in the CREATE VIEW statement.





# Appendix E: Results of Using ALTER TABLE

---

The following table describes how using the SQL ALTER TABLE statement impacts the CA Datacom Datadictionary definition.

## **ALTER TABLE table-name**

Updates the definition of the specified table.

## **column-name**

Defines a column (FIELD entity-occurrence) in the table. The following are the significant FIELD entity-occurrence attributes defined:

- AFTER= \$LAST or the SQL name of the column it follows.
- AGR-SQLNAME= The SQL name of the table.
- AUTHID= The schema (AUTHID) of the table.
- CLASS=S
- ENTITY-NAME = The *column-name* specified in the statement (the same name as used for SQLNAME).
- LENGTH= The length of the column.
- PARENT=START
- SQLNAME= The *column-name* specified in the statement (the same name as used for ENTITY-NAME).

Additional attributes are defined based on the column definition syntax.

The column must be added to the end of the table.

### **datatype**

Defines the column's data type through the following FIELD entity-occurrence attributes:

- SEMANTIC-TYPE= If the column's data type is DATE, TIME, or TIMESTAMP, this attribute reflects SQL-DATE, SQL-TIME, or SQL-STMP, respectively.
- SIGN=N
- TYPE=B
  - length 4 for SQL-DATE
  - length 3 for SQL-TIME
  - length 10 for SQL-STMP
- TYPE-NUMERIC=C

### **DEFAULT literal**

For the FIELD entity-occurrence, defines a literal value in the VALUE attribute and sets the DEFAULT-INSERT attribute to O.

### **DEFAULT USER**

For the FIELD entity-occurrence, sets the DEFAULT-INSERT attribute to U.

### **DEFAULT SYSTEM USER**

For the FIELD entity-occurrence, sets the DEFAULT-INSERT attribute to S.

### **DEFAULT NULL**

For the FIELD entity-occurrence, leaves the NULL-INDICATOR attribute unchanged. Removes a previously specified numeric or literal default value, but does not alter the nullability (NULL-INDICATOR attribute) of a column.

### **NOT NULL**

For the FIELD entity-occurrence, sets the NULL-INDICATOR attribute to N. NOT NULL can be used when adding a column, but not when modifying a column.

**PRIMARY KEY**

Creates a key consisting of this column or columns and a constraint. The following are the significant KEY entity-occurrence attributes defined:

- AGR-SQLNAME= The SQL name of the table.
- AUTHID= The schema (AUTHID) of the table.
- DATACOM-ID= An available 3-digit ID selected by CA Datacom Datadictionary for uniqueness within the database.
- DATACOM-NAME= SQ followed by three digits selected by CA Datacom Datadictionary for uniqueness within the database (same name as used for the ENTITY-NAME).
- ENTITY-NAME= SQ followed by three digits selected by CA Datacom Datadictionary for uniqueness within the database (same name as used for the DATACOM-NAME).
- FOREIGN=N
- INCLUDE-NIL-KEY=Y
- MASTER-KEY=N if not already Y
- MAX-KEY-LENGTH=0
- NATIVE-KEY=N if not already Y
- SQLNAME= The key's DATACOM-NAME followed by an underscore character followed by the concatenation of the DATACOM-NAME of the table and the DATACOM-ID of the database in which the key is defined (for example, SQ032\_INV00016).
- UNIQUE=Y

The following are the significant CONSTRAINT entity-occurrence attributes defined:

- AUTHID= The schema (AUTHID) associated with the table.
- ENTITY-NAME= An internally generated unique name consisting of the schema (the AUTHID used when creating the table) followed by a hyphen and the SQLNAME.
- SQLNAME= The name given to the constraint in the statement or an internally created name in the form CONSTRAINT\_ *nnnn* where the *nnnn* are four digits selected for uniqueness within the database.

The TABLE entity-occurrence has the following attributes that pertain to constraints:

- CNS-UNIQUE=Y
- CONSTRAINT=Y

You cannot copy over or delete a table or any part of a table while it has a constraint.

### UNIQUE

Defines a key consisting of the identified column or columns and a constraint. The following are the significant KEY entity-occurrence attributes defined:

- AGR-SQLNAME= The SQL name of the table.
- AUTHID= The schema (AUTHID) of the table.
- DATACOM-ID= An available 3-digit ID selected by CA Datacom Datadictionary for uniqueness within the database.
- DATACOM-NAME= SQ followed by three digits selected by CA Datacom Datadictionary for uniqueness within the database (same name as used for the ENTITY-NAME).
- ENTITY-NAME= SQ followed by three digits selected by CA Datacom Datadictionary for uniqueness within the database (same name as used for the DATACOM-NAME).
- FOREIGN=N
- INCLUDE-NIL-KEY=Y
- MASTER-KEY=N if this is not the first unique key of the table and a primary key is not defined; otherwise, MASTER-KEY=Y.
- MAX-KEY-LENGTH=0
- NATIVE-KEY=N if this is not the first unique key of the table and a primary key is not defined; otherwise, NATIVE-KEY=Y.
- SQLNAME= The key's DATACOM-NAME followed by an underscore character followed by the concatenation of the DATACOM-NAME of the table and the DATACOM-ID of the database in which the key is defined (for example, SQ032\_INV00016).
- UNIQUE=Y

The following are the significant CONSTRAINT entity-occurrence attributes defined:

- AUTHID= The schema (AUTHID) associated with the table.
- ENTITY-NAME= An internally generated unique name consisting of the schema (the AUTHID) used when creating the table followed by a hyphen and the SQLNAME.
- SQLNAME= The name given to the constraint in the statement or an internally created name in the form CONSTRAINT\_ *nnnn* where the *nnnn* are four digits selected for uniqueness within the database.

The TABLE entity-occurrence has the following attributes that pertain to constraints:

- CNS-UNIQUE=Y
- CONSTRAINT=Y

**WITH DEFAULT**

For the FIELD entity-occurrence, sets the DEFAULT-INSERT attribute to D.

**REFERENCES table-name**

Sets the TABLE entity-occurrence CNS-REFERS= and CONSTRAINT= attributes to Y for this table.

Sets the referenced TABLE entity-occurrence CNS-REFERENCED= and CONSTRAINT= attributes to Y.

**CHECK (search condition)**

Creates a constraint. The following are the significant CONSTRAINT entity-occurrence attributes defined:

- AUTHID= The schema (AUTHID) associated with the table.
- ENTITY-NAME= An internally generated unique name consisting of the schema (the AUTHID used when creating the table) followed by a hyphen and the SQLNAME.
- SQLNAME= The name given to the constraint in the statement or an internally created name in the form CONSTRAINT\_ *nnnn* where the *nnnn* are four digits selected for uniqueness within the database.

The TABLE entity-occurrence has the following attributes that pertain to constraints:

- CNS-DOMAIN=Y
- CONSTRAINT=Y

**CONSTRAINT constraint-name**

Sets the CONSTRAINT entity-occurrence SQLNAME attribute to the name supplied in this parameter.

**ADD table constraint**

Modifies the table to add one of the constraints that follows to the table.

**FOREIGN KEY (column-list)**

Defines a key consisting of this column or columns and a constraint. The KEY entity-occurrence has the following attributes:

- AGR-SQLNAME= The SQL name of the table.
- AUTHID= The schema (AUTHID) of the table.
- DATACOM-NAME= blanks
- DATACOM-ID=000

- ENTITY-NAME= CA Datacom Datadictionary uses the constraint's ENTITY-NAME value.
- FOREIGN=Y
- INCLUDE-NIL-KEY=Y
- MASTER-KEY=N
- MAX-KEY-LENGTH=00
- NATIVE-KEY=N
- SQLNAME= CA Datacom Datadictionary uses the constraint's SQLNAME value.
- UNIQUE=N

The following are the significant CONSTRAINT entity-occurrence attributes defined:

- AUTHID= The schema (AUTHID) associated with the table.
- ENTITY-NAME= An internally generated unique name consisting of the schema (the AUTHID) used when creating the table followed by a hyphen and the SQLNAME.
- SQLNAME= The name given to the constraint in the statement or an internally created name in the form CONSTRAINT\_ *nnnn* where the *nnnn* are four digits selected for uniqueness within the database.

The TABLE entity-occurrence has the following attributes:

- If all tables referred to by the foreign key are in the same base, CNS-SAME-BASE=Y; otherwise, CNS-SAME-BASE=N.
- CNS-REFERS=Y
- CONSTRAINT=Y

Sets the referenced TABLE entity-occurrence CNS-REFERENCED= and CONSTRAINT= attributes to Y.

**REFERENCES table-name (column-list)**

Sets the TABLE entity-occurrence CNS-REFERS= and CONSTRAINT= attributes to Y for this table.

Sets the referenced TABLE entity-occurrence CNS-REFERENCED= and CONSTRAINT= attributes to Y.

**CHECK (search condition)**

Creates a constraint. The following are the significant CONSTRAINT entity-occurrence attributes defined:

- AUTHID= The schema (AUTHID) associated with the table.
- ENTITY-NAME= An internally generated unique name consisting of the schema (the AUTHID used when creating the table) followed by a hyphen and the SQLNAME.
- SQLNAME= The name given to the constraint in the statement or an internally created name in the form CONSTRAINT\_ *nnnn* where the *nnnn* are four digits selected for uniqueness within the database.

**CONSTRAINT constraint-name**

Sets the CONSTRAINT entity-occurrence SQLNAME attribute to the name supplied in the statement.

**DROP CONSTRAINT constraint-name**

Obsoletes the CONSTRAINT entity-occurrence specified. The KEY entity-occurrence remains. Views that reference the constraint's table are marked invalid.

You can drop the foreign key only if there are no primary keys that reference it. Views based on this table are marked invalid.

**DROP column-name**

Obsoletes the column. All plans, views, and synonyms that reference the column are marked invalid.

**DROP PRIMARY KEY**

Obsoletes the CONSTRAINT entity-occurrence. The KEY entity-occurrence remains. The DROP PRIMARY KEY statement is successful only if there are no foreign keys that reference this primary key. Views based on the table associated with the constraint are marked as invalid.

**DROP FOREIGN KEY identifier**

Obsoletes the KEY entity-occurrence created as the foreign key and the CONSTRAINT entity-occurrence.

Views based on the table associated with the obsoleted key are marked as invalid.

**MODIFY column-name DEFAULT literal**

For the FIELD entity-occurrence, defines a literal value in the VALUE attribute and sets the DEFAULT-INSERT attribute to O.

**MODIFY column-name DEFAULT USER**

For the FIELD entity-occurrence, sets the DEFAULT-INSERT attribute to U.

**MODIFY column-name DEFAULT SYSTEM USER**

For the FIELD entity-occurrence, sets the DEFAULT-INSERT attribute to S.

**MODIFY column-name DEFAULT NULL**

For the FIELD entity-occurrence, sets the DEFAULT-INSERT attribute to N.

**MODIFY column-name data type**

Modifies the column's data type changing the following FIELD entity-occurrence attributes:

- SEMANTIC-TYPE= If the column's data type is changed from DATE, TIME, or TIMESTAMP to another data type, this attribute reflects blanks.
- SIGN= N if character, Y if numeric.
- TYPE= becomes the new data type you specify.
- TYPE-NUMERIC= conventional.

All plans and views that reference the column are marked invalid.

**RENAME column-name TO column-name**

For the FIELD entity-occurrence, changes the SQLNAME to the name specified in the RENAME statement. All plans, views, and synonyms that reference the column are marked invalid. The ENTITY-NAME attribute is not changed from its current value in CA Datacom Datadictionary.



# Appendix F: SQL Object Consistency Analyzer and Upgrade Rebind Utilities

---

Discussed in this chapter are the SQL Object Consistency Analyzer (see [SQL Object Consistency Analyzer](#) (see page 849)) and the SQL Upgrade Rebind Utility (see [DBSRFPR \(SQL Upgrade Rebind Utility\)](#) (see page 861)) which, beginning in r10, can also be used to drop plans.

## SQL Object Consistency Analyzer

Each SQL object consists of information stored in various tables managed by SQL and CA Datacom Datadictionary. A successful rebind depends on the accessibility of that information. The CA Datacom/DB SQL Upgrade Rebind Utility cannot execute successfully if the SQL objects (constraints, plans, and views) that it rebinds are not intact in the CA Datacom/DB system to be rebound. The SQL Object Consistency Analyzer allows you to verify that the needed information is accessible. Therefore, before beginning an upgrade to a new CA Datacom/DB version, execute the SQL Object Consistency Analyzer against your current CA Datacom/DB system. If any problems are discovered, correct them before you begin the upgrade.

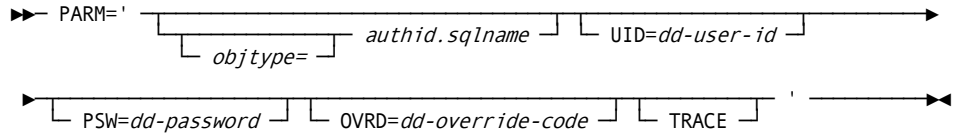
**Note:** You can also execute the SQL Object Consistency Analyzer against the new CA Datacom/DB version system if you suspect rows are missing in SQL-managed tables (in the areas in the DATA-DICT and DDD-DATABASE databases).

The information on the SQL Object Consistency Analyzer is divided into three sections:

- Running the SQL Object Consistency Analyzer (see [Running the SQL Object Consistency Analyzer](#) (see page 850)).
- Interpreting the Output (see Sample Report).
- Correcting Problems (see [Correcting Problems](#) (see page 856)).

## Running the SQL Object Consistency Analyzer

The SQL Object Consistency Analyzer can verify all constraints, views, and plans in a single execution, or can be limited to a subset. Analysis can be limited by object type, authorization ID, and object name as indicated by the following syntax diagram. This syntax should be added as a parameter to the execution statement in your JCL. **The parameter may be omitted** if all constraints, views, and plans are to be analyzed and the default values for UID=, PSW=, and OVRD= are acceptable.



**Note:** Use a single space to separate parameters, but do *not* use any other spaces (or any commas) in the statement. The parameters can be entered in any order. Use uppercase letters to specify values for the parameters.

### **objtype=**

(Optional) Identifies the entity-type of the object(s) to be verified.

#### **Valid Entries:**

CONSTRAINT= or PLAN= or VIEW=

#### **Default:**

If *objtype=* is not specified, constraints, plans, and views are verified.

### **authid**

(Required, if *objtype=* is specified.) Enter the authorization ID of object(s) to be verified, or \* (an asterisk) to verify all AUTHIDs of the specified entity type(s).

#### **Valid Entries:**

An existing AUTHID or \* (the AUTHID may not exceed 18 characters.)

#### **Default:**

(No default)

### **.sqlname**

(Required, if *authid* is specified.) Enter the name of a constraint, view, or plan, or \* (an asterisk). Specifying \* will cause all entities of the specified AUTHID to be verified. If \* was specified as the AUTHID, then SQL name is treated as if \* was specified.

#### **Valid Entries:**

The SQL name of an existing constraint, view, or plan, or \* (the name may not exceed 32 characters).

#### **Default:**

(No default)

**UID=**

*(Optional)* You must specify a valid user ID if DATACOM-INSTALL has been removed as a valid CA Datacom Datadictionary user ID (a PERSON entity-occurrence), or if NEWUSER has been removed as the CA Datacom Datadictionary password for the DATACOM-INSTALL user ID.

**Note:** If you specify a CA Datacom Datadictionary user ID but not a CA Datacom Datadictionary password, blanks are used as the password.

**Valid Entries:**

A valid CA Datacom Datadictionary user ID

**Default:**

DATACOM-INSTALL

**PSW=**

*(Optional)* You must specify a valid password assigned to the CA Datacom Datadictionary user ID if DATACOM-INSTALL has been removed as a valid CA Datacom Datadictionary user ID (a PERSON entity-occurrence), or if NEWUSER has been removed as the DATACOM-INSTALL password, or if UID= has been specified but the password is something other than blanks. If the password parameter is specified, then the user ID must also be specified.

**Valid Entries:**

A valid CA Datacom Datadictionary password

**Default:**

blanks (if UID= is specified) or  
NEWUSER (if UID= is not specified)

**OVRD=**

*(Optional)* A valid four-character CA Datacom Datadictionary override code must be specified if PRIV is no longer the CA Datacom Datadictionary override code.

**Valid Entries:**

A valid four-character CA Datacom Datadictionary override code

**Default:**

PRIV

**TRACE**

(Optional) Prints additional output to help CA Support diagnose problems found by the Analyzer.

**Important!** Do not specify any entity-type that includes the character string TRACE unless instructed to do so by CA Support.

**Valid Entries:**

TRACE

**Default:**

(No default)

## Sample JCL

Following are JCL examples showing the limiting of analysis to a specific entity-type (and a certain AUTHID) and showing the analysis of all constraints, views, and plans. For more information about the target libraries, see Listing Libraries for CA Datacom Products.

**Note:** These JCL examples contain lowercase letters indicating entries you must replace with code that meets your installation and site standards.

```
//jobname JOB (acctinfo),'submitter name',CLASS=A,MSGCLASS=X,REGION=2048K
//SQLEXEC1 EXEC PGM=DBSOCPR,COND=(0,NE),REGION=1024K,
//      PARM='PLAN=authid.*'
//STEPLIB DD DSN=cai.db90.svclib,DISP=SHR      CA Datacom SVC library
//      DD DSN=cai.db90.loadlib,DISP=SHR      CA Datacom load library
//      DD DSN=cai.db100.cuslib,DISP=SHR      CA Datacom custom library
//      DD DSN=cai.db100.CAILIB,DISP=SHR      CA Datacom target library
//      DD DSN=cai.ca90s.wu40.CAILIB,DISP=SHR VPE target library
//SYSUDUMP DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
/*
//jobname JOB (acctinfo),'submitter name',CLASS=A,MSGCLASS=X,REGION=2048K
//SQLEXEC1 EXEC PGM=DBSOCPR,COND=(0,NE),REGION=1024K,
//      PARM='*.*'
//STEPLIB DD DSN=cai.db90.svclib,DISP=SHR      CA Datacom SVC library
//      DD DSN=cai.db90.loadlib,DISP=SHR      CA Datacom load library
//      DD DSN=cai.db100.cuslib,DISP=SHR      CA Datacom custom library
//      DD DSN=cai.db100.CAILIB,DISP=SHR      CA Datacom target library
//      DD DSN=cai.ca90s.wu40.CAILIB,DISP=SHR VPE target library
//SYSUDUMP DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
/*
```

Ensure that your new version's library is concatenated behind your previous version's library.

```
* $$ JOB JNM=SQLENT,CLASS=n,USER='CAINS'
* $$ LST CLASS=a,DEST=(*,vmuidnn)
// JOB SQLENT ANALYSIS BY ENTITY-TYPE
// ASSGN SYS025,DISK,VOL=volser,SHR
// DLBL db90,'db90.customer.library'
// EXTENT SYS025,volser
// DLBL db100,'db100.customer.library'
// EXTENT SYS025,volser
// DLBL vpe,'cai.ca90s.wu40.library'
// EXTENT SYS025,volser
// LIBDEF *,SEARCH=(db90.db90,db100.db100,vpe.vpe,ca90s.ca90s)
/*
/&
* $$ E0J
* $$ JOB JNM=SQLALL,CLASS=n,USER='CAINS'
* $$ LST CLASS=a,DEST=(*,vmuidnn)
// JOB SQLALL ANALYSIS OF ALL CONSTRAINTS, VIEWS, AND PLANS
// ASSGN SYS025,DISK,VOL=volser,SHR
// DLBL db90,'db90.customer.library'
// EXTENT SYS025,volser
// DLBL db100,'db100.customer.library'
// EXTENT SYS025,volser
// DLBL vpe,'cai.ca90s.wu40.library'
// EXTENT SYS025,volser
// LIBDEF *,SEARCH=(db90.db90,db100.db100,vpe.vpe,ca90s.ca90s)
// EXEC DBSOCPR,SIZE=1024K,PARM='*.*'
/*
/&
* $$ E0J
```

## General Utility Sample Report

Following are two pages of a General Utility report. For a sample header for this report, see Sample Report Headers.

```
CONSTRAINT REBIND SUCCESSFUL FOR TABLE sysadm.testplan1
CONSTRAINT REBIND SUCCESSFUL FOR TABLE sysadm.testplan2
CONSTRAINT REBIND SUCCESSFUL FOR TABLE sysadm.testplan3
SUCCESSFULLY REBOUND VIEW sysadm.testview1
SUCCESSFULLY REBOUND VIEW sysadm.testview2
SUCCESSFULLY REBOUND VIEW sysadm.testview3
SUCCESSFULLY REBOUND PLAN sysadm.testplan1
REBIND FAILED FOR PLAN sysadm.testplan2 - SEE PXX FOR ERRORS
SUCCESSFULLY REBOUND PLAN sysadm.testplan3
SUCCESSFULLY REBOUND PLAN sysadm.testplan4
```

```
**** RELEASE BOUNDARY UPGRADE REBIND SUMMARY ****
*
***** CONSTRAINTS *****
*
* CONSTRAINED TABLES ALREADY REBOUND      : nnnnnnnn *
* CONSTRAINED TABLES SUCCESSFULLY REBOUND : nnnnnnnn *
* CONSTRAINED TABLES WITH REBIND ERRORS   : nnnnnnnn *
* TOTAL CONSTRAINED TABLES                 : nnnnnnnn *
*
***** VIEWS *****
*
* VIEWS          ALREADY REBOUND           : nnnnnnnn *
* VIEWS          REBOUND                    : nnnnnnnn *
* VIEWS          WITH REBIND ERRORS        : nnnnnnnn *
* TOTAL VIEWS                                         : nnnnnnnn *
*
***** PLANS *****
*
* PLANS          ALREADY REBOUND           : nnnnnnnn *
* PLANS          REBOUND                    : nnnnnnnn *
* PLANS          WITH REBIND ERRORS        : nnnnnnnn *
* TOTAL PLANS                                         : nnnnnnnn *
*
*****
```

## Object Consistency Analyzer Sample Report

The following is an example of the output produced by the SQL Object Consistency Analyzer. If the summary for each entity-type indicates that all entities were found intact, then your task is complete. Otherwise, use the information printed in the report to correct the problem as described in [Correcting Problems](#) (see page 856). In the following report, note that the name and type of the SQL object involved is printed for each problem found, as well as a specific statement or statements indicating which pieces of that entity-type are missing:

For a sample header for this report, see Sample Report Headers.

```
ECHO PRINT OF PARMS:
WINDIGO.*.

RESULTS OF CONSTRAINT CONSISTENCY CHECKING:

ERRORS FOUND IN TABLE WINDIGO.ORDER CONSTRAINT MIN_PMT AS FOLLOWS:
CONSTRAINT IS MISSING FROM DATADITIONARY.

CONSTRAINTS DEFINED ON THE FOLLOWING DATABASES WILL NOT REBIND
DUE TO DD STRUCTURE VERIFY ERRORS:

DB ID   DB NAME                DD ERROR/MODULE  DESCRIPTION
-----
  481   TEST-DATABASE         SV4 /            STRUCTURE VERIFY ERROR

NOTE: DD STRUCTURE VERIFY ERRORS NOT INCLUDED IN NUMBERS REPORTED IN FOLLOWING
TOTAL CONSTRAINTS PROCESSED: 118
VALID CONSTRAINTS:          117
INVALID CONSTRAINTS:         1
-----

RESULTS OF PLAN CONSISTENCY CHECKING:

ERRORS WERE FOUND IN PLAN WINDIGO.XYZ AS FOLLOWS:
PLAN IS MISSING FROM DD TABLE PLN.
STATEMENT(S) ARE MISSING FROM TABLES DDD AND/OR STM.

TOTAL PLANS PROCESSED: 281
VALID PLANS:          280
INVALID PLANS:         1
-----

RESULTS OF VIEW CONSISTENCY CHECKING:

ALL VIEWS WERE FOUND INTACT.

TOTAL VIEWS PROCESSED: 17
VALID VIEWS FOUND:    17
INVALID VIEWS FOUND:   0
-----
```

You could also receive error messages indicating problems that prevented DBSOCPR from running normally. Most of the messages are self explanatory and include instructions for correcting the problem. Following are messages that may be more difficult to interpret:

**DB RETURN CODE xx(yyy DEC) PRODUCED BY zzzzz COMMAND. CONSISTENCY CHECKING IS BEING ABORTED.**

If you receive this message, xx is the CA Datacom/DB external return code, yyy is the internal return code in decimal, and zzzzz is the record-at-a-time command that produced the error. Look up the return code in the *CA Datacom/DB Message Reference Guide*, correct the problem as described in that manual, and retry the SQL Object Consistency Analyzer.

**DD USRINITI FAILED. RC=return-code, ERROR=dd-error-message,USER=uid,password PLEASE SPECIFY OR CHECK THE UID AND PSW PARAMETERS.**

If you receive this message, first follow the printed suggestion to check your UID and PSW parameters for correctness (see the PARM= documentation on the preceding pages). If no problem is found, check for a message in the job output (JES LOG for z/OS) indicating that a requested module was not found. If no problem is found here, contact CA Support for a support person to look at your RC, ERROR, and USER information printed in the error message to determine what the problem is. Only contact CA Support if the first two steps were unproductive.

## Correcting Problems

Correcting problems found by the SQL Object Consistency Analyzer involves deleting the invalid data and restoring the constraint, plan, or view to its original condition.

**Important!** First make backups of the following areas:

- DDD (Data Definition Directory)
- SIT (Schema Information Tables)
- CXX (CA Datacom/DB Directory)
- DD (CA Datacom Datadictionary )

If you no longer need the constraint, plan, or view in question, you can skip the steps that involve saving the source and re-creating the constraint, plan, or view. All reports and documentation produced should be saved until successful re-creation or deletion of the constraint, view, or plan has been achieved. If you call CA Support during this process, have the necessary documentation ready for their use.



The remainder of this section is divided into three areas:

- Correcting Constraint Problems (see [Correcting Constraint Problems](#) (see page 857))
- Correcting Plan Problems (see [Correcting Plan Problems](#) (see page 859))
- Correcting View Problems (see [Correcting View Problems](#) (see page 860))

## Correcting Constraint Problems

Read all instructions before attempting any of the steps outlined in this section, and if any step is unclear, call CA Support.

Constraint problems cause the following message to be generated:

**ERRORS FOUND IN TABLE authid.tablename CONSTRAINT constraint-name AS FOLLOWS:**

Following that message will be other messages detailing the problems that were found. Here is a list of possible messages:

**CONSTRAINT IS MISSING FROM DATADictionary**

**CONSTRAINT IS MISSING FROM SYSADM.SYSCONSTROBJ**

**CONSTRAINT IS MISSING FROM SYSADM.SYSCONSTRSRC**

**CONSTRAINT IS MISSING FROM SYSADM.SYSCONSTRDEP**

If any of these messages appear, use the CA Datacom Datadictionary Interactive SQL Service Facility to accomplish the following steps:

1. Obtain the SQL statement that created the constraint using one of the following:
  - Obtain the CREATE TABLE or ALTER TABLE SQL statement that was used to create the constraint in question. If this statement is not available, run the following query to obtain the constraint source:

```
SELECT TEXT, SEQNO FROM SYSADM.SYSCONSTRSRC WHERE
      CNAME='constraint-name' AND CREATOR='authid' ORDER BY SEQNO.
```
  - If the SELECT above was unsuccessful, run a DDUTILITY Text Report on the CONSTRAINT entity-occurrence, specifying a text CLASS of SQLSOURCE. (See the *CA Datacom Datadictionary Batch Reference Guide* for details on running this report.)

- Issue the following SQL statement to drop the constraint:

```
ALTER TABLE authid.table-name DROP CONSTRAINT constraint-name.
```

- If the ALTER TABLE succeeds, go to step 4. If the ALTER TABLE does not succeed:

- Execute the following SQL DELETE statements to remove traces of the constraint from the SIT area.

```
DELETE FROM SYSADM.SYSCONSTRSRC WHERE CNAME='constraint-name' AND  
CREATOR='authid'
```

```
DELETE FROM SYSADM.SYSCONSTRDEP WHERE CNAME='constraint-name' AND  
CCREATOR='authid' (CCREATOR is not a misprint)
```

```
DELETE FROM SYSADM.SYSCONSTROBJ WHERE CNAME='constraint-name' AND  
CREATOR='authid'
```

- Execute a DDUTILITY Index Report on constraints to determine if traces of the constraint exist.

- If no traces are found, go to step 4. If traces are found, use DDUPDATE to delete the CONSTRAINT entity-occurrence. If you require assistance, call CA Support.

- Re-add the constraint by running an SQL statement in the form:

```
ALTER TABLE authid.tablename ADD insert-constraint-text-here  
CONSTRAINT constraint-name.
```

- If the ALTER TABLE succeeds, your problem has been solved. If the ALTER TABLE fails, consult the *CA Datacom/DB Message Reference Guide* to determine the appropriate action to take for the SQLCODE you received. After remedying the problem that caused the nonzero SQLCODE, retry the ALTER TABLE.

If you receive this message:

**CONSTRAINTS DEFINED ON THE FOLLOWING DATABASES WILL NOT REBIND DUE TO DD STRUCTURE VERIFY ERRORS:**

- Note the DD ERROR, MODULE, and DESCRIPTION listed for each database that had an error.
- Use the *CA Datacom/DB Message Reference Guide* to find out what went wrong.
- If the information printed in the report does not give you enough information to find the problem, perform the CA Datacom Datadictionary VERIFY function on the database in question and correct the problems indicated by the output from the function.
- After problems are corrected, run DBSOCPR again to confirm that the problems no longer exist.

## Correcting Plan Problems

Read all instructions before attempting any of the steps outlined in this section, and if any step is unclear, call CA Support.

Plan problems cause the following message to be generated:

**ERRORS WERE FOUND IN PLAN authid.plan-name AS FOLLOWS:**

This message is followed by further messages detailing the problems that were found. The following is a list of possible messages and the corrective actions they require:

**PLAN IS MISSING FROM DD TABLE PLN**

**STATEMENT(S) ARE MISSING FROM TABLES DDD AND/OR STM**

**PLAN IS INVALID — REBIND BEFORE RUNNING DBSRFPR**

If either of the first two messages above are received, attempt to drop the plan using the DELETE PLAN administrative task in the CA Datacom Datadictionary Interactive SQL Service Facility.

If the DELETE PLAN succeeds, restore the plan by re-preparing (re-precompiling) the program that generated the plan. The problem is then solved.

If the DELETE PLAN or the re-prepare fails:

1. Execute the DDDDULM utility with FUNC=INDEX. (See *CA Datacom/DB Database and System Administration Guide* for instructions on using the DDDDULM utility.) Look for entries beginning with PLN, SRC, PRP, and DEP, followed immediately by the AUTHID and the SQL name of the plan (see the report headers).
2. Execute DDUTILTY Index Reports on the PLAN and STATEMENT entity-types, looking for names consisting of the plan's AUTHID, a dash, then the *sqlname*. The statements contain the plan name followed by the statement number in numeric form. For example, statement one for plan SYSUSR.TESTPLAN would appear as SYSUSR-TESTPLAN0000000000000001. If you require assistance in deleting a PLAN entity-occurrence, or if the report shows that some other combination of circumstances exist, or if you need assistance interpreting the report, call CA Support.
3. After the plan has been successfully deleted, restore the plan by re-preparing (re-precompiling) the program that generated it.

Rebind the specified plan if you receive the message:

**PLAN IS INVALID — REBIND BEFORE RUNNING DBSRFPR**

## Correcting View Problems

Read all instructions before attempting any of the steps outlined in this section, and if any step is unclear, call CA Support.

View problems cause the following message to be generated:

**ERROR(S) WERE FOUND IN VIEW authid.view-name AS FOLLOWS:**

This message is followed by other messages detailing the problems that were found. The following is a list of other messages that might appear:

**VIEW IS MISSING FROM DATADictionary**

**THE DDD IS MISSING ROWS FOR THIS VIEW**

If either of these messages appears, take the following steps:

1. Locate your original CREATE VIEW statement.
2. Drop the view using Interactive SQL Service Facility to execute the SQL statement:  

```
DROP VIEW authid.viewname
```
3. If the DROP fails, go to the following step 4. If the DROP succeeds:
  - a. Re-create the view using your CREATE VIEW source.
  - b. If the CREATE VIEW fails, consult the *CA Datacom/DB Message Reference Guide* for instructions on recovering from your particular SQLCODE, then correct the problem that caused the SQLCODE and retry the CREATE VIEW.

**Note:** The problem is solved when the CREATE VIEW succeeds.

4. When the DROP fails:
  - a. Execute the three SQL DELETE statements detailed in the *Correcting Constraint Problems* section that begins on [Correcting Constraint Problems](#) (see page 857), substituting the view name for the constraint-name and ignoring any 100 SQLCODEs received.
  - b. Use the Interactive SQL Service Facility to execute the following SQL statement:  

```
DELETE FROM SYSADM.SYSVIEWDEP WHERE  
DCREATOR='authid' and DNAME='view-name'.
```
  - c. Note any negative SQL return code received. See the *CA Datacom/DB Message Reference Guide* for information about specific return codes.
  - d. Execute the DDDDULM utility with FUNC=INDEX. Look for entries beginning with SRC, PRP, COL, REL, and DEP, followed immediately by the AUTHID and the SQL name of the view (see the report headers).
  - e. Execute a DDUTILTY Index Report on the VIEW entity-type, looking for names consisting of the AUTHID followed by the view name, separated by a dash.

- f. If no traces of the view are found by the DDDDULM report but the Index Report shows there is a VIEW entity-occurrence in CA Datacom Datadictionary, use DDUPDATE to delete the VIEW entity-occurrence. If you need assistance, or if the reports show that some other combination of circumstances exist, or if you require help interpreting the reports, call CA Support.
- g. Go to step 3 and follow the instructions listed for when the DROP succeeds.

## DBSRFPR (SQL Upgrade Rebind Utility)

A restartable utility program, DBSRFPR, gives you an easy way to do common tasks such as rebinding plans, views, and constraints (see the following). Starting in r10, DBSRFPR can also be used to drop plans (see [Dropping Plans with DBSRFPR](#) (see page 865)).

### Rebinding with DBSRFPR

When upgrading to a new CA Datacom/DB version, all constraints, plans, and views of the previous version must be rebound, but it is not necessary that all constraints, plans, and views be successfully rebound during the upgrade process. You can begin using the new version for the objects rebound and correct remaining rebind problems at a later time. If an object is accessed before it is upgraded, an automatic rebind is attempted.

DBSRFPR can be used to force the rebinding of SQL objects. If desired, you can use this facility after applying solutions that correct errors in bound objects. (The solution's USER ACTION REQUIRED will state that affected objects, usually plans, must be rebound.) Or, for performance reasons, you can use this facility to rebind plans instead of using the CA Datacom Datadictionary Interactive SQL Service Facility plan rebind facility.

Specifying REBIND forces rebinds to be performed even on items that have not been marked *invalid* by CA Datacom Datadictionary. Valid items whose rebinds are forced in this way are reported as **SUCCESSFULLY REBOUND** instead of **ALREADY REBOUND**.

To rebind objects already upgraded or select specific objects to be rebound, add the following parameter to the execution statement in your JCL.

```

▶▶ PARM= ' _____
           |_____|
           |  objtype=  |  authid.sqlname  |  REBIND  _____
           |_____|                |  CHECKONLY  |  _____
           |_____|
           |  MSG=xy  |  '
  
```

**Note:** Use a single space to separate parameters, but do *not* use any other spaces (or any commas) in the statement. The parameters can be entered in any order. Use uppercase letters to specify values for the parameters.

**objtype=**

*(Optional)* Identifies the entity-type of the object(s) to be rebound.

With regard to specifying CONSTRAINT= for *objtype=*, the table name is used rather than a constraint name because all constraints attached to a table are invalidated at the same time, and all constraints need to be rebound before the table is fully useable. Therefore, to save steps and prevent mistakes specify for the value of CONSTRAINT= the table name to which the constraint belongs.

**Valid Entries:**

CONSTRAINT= or PLAN= or VIEW=

**Default:**

If *objtype=* is not specified, constraints, plans, and views are rebound.

**authid**

*(Required, if objtype= is specified.)* Enter the authorization ID of object(s) to be rebound, or \* (an asterisk) to rebind all AUTHIDs of the specified entity-type(s).

**Valid Entries:**

An existing AUTHID or \* (the AUTHID may not exceed 18 characters.)

**Default:**

(No default)

**.sqlname**

*(Required, if authid is specified.)* Enter the name of a table, view, or plan, or \* (an asterisk). Specifying \* will cause all entities of the specified AUTHID to be rebound. You can rebind a given entity name under every AUTHID by specifying an asterisk for authid and a specific *sqlname*, that is, *objtype=\*.sqlname*.

**Valid Entries:**

The SQL name of an existing table, view, or plan, or \* (the name may not exceed 32 characters).

**Default:**

(No default)

**REBIND**

*(Optional)* Use after upgrade to rebind an object of the new version.

**CHECKONLY**

*(Optional)* When CHECKONLY is specified, CA Datacom/DB reports which plans have bad options and which have bad SQL version/release numbers, but no rebinds are performed. CHECKONLY is ignored if REBIND has been specified (this exclusivity is shown in the previous syntax diagram as a choice between the two).

The plan summary at the end of PARM= execution is different when CHECKONLY is specified in that the number of plans with mismatched versions, bad options, and the total number of plans is reported. If there were plans that had bad options and mismatched versions, those two numbers add to a total that can therefore have a sum greater than the number of TOTAL PLANS displayed. Messages are displayed for plans with bad options and/or mismatched versions.

**MSG=xy**

*(Optional)* Specify the MSG plan option when rebinding plans. The x refers to precompile-time messages and y refers to messages generated by the Optimizer when the statement is executed. Use S to specify summary, D to specify detail, or N to specify none.

**Valid Entries:**

SS, DD, SD, DS, NS, ND, DN, SN, or NN

**Default Value:**

The default is the MSG plan option for the existing plan in the DDD. If MSG= is specified, it overrides the specification in the existing plan.

Because the following areas are updated, you should back them up before running DBSRFPR:

- CA Datacom/DB Directory (CXX)
- Data Definition Directory (DDD)
- CA Datacom Datadictionary (DD)
- SQL Information Tables (SIT)
- SYSADM.SYSMSG table (MSG)

**Order in Which Objects Are Rebound**

Objects are rebound in the following order:

1. Constraints
2. Views
3. Plans

A success or failure message in the following formats is written for each object.

```

CONSTRAINT REBIND ALREADY DONE FOR TABLE xxxxxxxxxxxxxxxxxxxxxxxxxxxx
CONSTRAINT REBIND SUCCESSFUL FOR TABLE xxxxxxxxxxxxxxxxxxxxxxxxxxxx
CONSTRAINT REBIND **FAILED** FOR TABLE xxxxxxxxxxxxxxxxxxxxxxxxxxxx
... xxxxxxxxxxxxxxxxxxxxxxxxxxxx NOT REBOUND
... xxxxxxxxxxxxxxxxxxxxxxxxxxxx SHOULD HAVE BEEN REBOUND

ALREADY REBOUND VIEW xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
REBIND FAILED FOR VIEW xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
SUCCESSFULLY REBOUND VIEW xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

ALREADY REBOUND PLAN xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
REBIND FAILED FOR PLAN xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
SUCCESSFULLY REBOUND PLAN xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    
```

If execution is prematurely ended due to a fatal error, such as table full or index full, the following message is issued:

```

**** FATAL ERROR - UPGRADE INCOMPLETE ****
    
```

A summary report shows, by object type, the number of successful and unsuccessful rebinds, and previously rebound objects:

```

***** VERSION BOUNDARY UPGRADE REBIND SUMMARY *****
*                                                                 *
***** CONSTRAINTS *****
*                                                                 *
* CONSTRAINED TABLES ALREADY REBOUND      : nnnnnnnnn *
* CONSTRAINED TABLES SUCCESSFULLY REBOUND : nnnnnnnnn *
* CONSTRAINED TABLES WITH REBIND ERRORS   : nnnnnnnnn *
* TOTAL CONSTRAINED TABLES                : nnnnnnnnn *
*                                                                 *
***** VIEWS *****
*                                                                 *
* VIEWS          ALREADY REBOUND      : nnnnnnnnn *
* VIEWS          REBOUND              : nnnnnnnnn *
* VIEWS          WITH REBIND ERRORS   : nnnnnnnnn *
* TOTAL VIEWS                    : nnnnnnnnn *
*                                                                 *
***** PLANS *****
*                                                                 *
* PLANS          ALREADY REBOUND      : nnnnnnnnn *
* PLANS          REBOUND              : nnnnnnnnn *
* PLANS          WITH REBIND ERRORS   : nnnnnnnnn *
* TOTAL PLANS                    : nnnnnnnnn *
*****
    
```



Diagnostic information to correct rebind failures is found in the Statistics and Diagnostics Area (PXX). You might need to allocate a larger than normal PXX data set.

The upgrade rebind utility is restartable. Each time it is executed it finds the objects not yet rebound. It issues COMMIT WORK after each successful rebind, so it can be canceled at any time and restarted.

Although the new plans will normally be more efficient, optimization messages are written to the SYSADM.SYSMSG table to help resolve any performance problems.

- Before running the upgrade utility, you should back up or copy the SYSADM.SYSMSG table to preserve the optimization messages of the previous version.
- If the plan option MSG is not used to request summary or detail bind-time messages, summary-level bind-time messages are forced during the rebind. You should ensure that this area is large enough to hold messages inserted by the upgrade utility. If this or any other area becomes full, the utility program will end.
- Special upgrade messages beginning with REBIND: indicate differences in join order or method, and when a sort is required when one was not previously required. See the SQL Optimization Messages section in the *CA Datacom/DB Message Reference Guide*.

You can access these messages for review with the following query:

```
SELECT *
FROM SYSADM.SYSMSG
WHERE SUBSTR(MSG, 1, 7) = 'REBIND:';
```

**Note:** These messages are deleted the first time the plan is rebound after being upgraded.

In most cases these changes will result in better efficiency, but you can force the join order of the previous version by placing the tables in the FROM clause in the join order of the previous version, setting plan option OPT=M, and recompiling the program. This also forces the nested loop join method.

## Dropping Plans with DBSRFPR

DBSRFPR can be used to drop plans. In order to use this feature, add the following parameter (PARM=) to the execution statement in your JCL. following is the syntax diagram for using the PARM= parameter of DBSRFPR to drop plans:

```
▶▶ PARM= ' ┌ DROP PLAN=authid.sqlname ────┐ ───────────────────────────────────▶
          └ DROP ALL PLAN=*. * ───────────┘
```

**DROP or DROP ALL**

Specify a drop command of either DROP or DROP ALL to indicate that you want a specified set of plans dropped from the MUF. Code DROP PLAN=*authid.sqlname* carefully to be certain you drop only the plans you want to drop. If you want every plan in your MUF dropped, use DROP ALL and PLAN=\*. \* (an asterisk followed by a period followed by another asterisk).

***authid***

*(Required.)* Enter the authorization ID of object(s) to be dropped, or \* (an asterisk) to drop all AUTHIDs of the specified entity type.

**Valid Entries:**

An existing AUTHID or \* (the AUTHID may not exceed 18 characters.)

**Default:**

(No default)

***.sqlname***

*(Required, if authid is specified.)* Enter the name of a plan, or \* (an asterisk). Specifying \* will cause all entities of the specified AUTHID to be dropped. If \* was specified as the AUTHID, then SQL name is treated as if \* was specified.

**Valid Entries:**

The SQL name of an existing plan, or \* (the name may not exceed 32 characters).

**Default:**

(No default)

## Sample JCL

The JCL contains lowercase letters indicating entries you must replace with code that meets your installation and site standards.

```
//jobname JOB acctinfo,'submitter name',MSGLEVEL=1
/*
/* NOTE: The above job statement is for example only.
/* Code the job statement according to your installation standards.
/*
//DBS81 EXEC PGM=DBSRFPR,REGION=2048K
//STEPLIB DD DSN=cai.db100.cuslib,DISP=SHR CA Datacom custom library
// DD DSN=cai.db100.CAILIB,DISP=SHR CA Datacom target library
// DD DSN=cai.ca90s.wu40.CAILIB,DISP=SHR VPE target library
//SYSPRINT DD SYSOUT=* Print Output
//SYSUDUMP DD SYSOUT=*
/*
* $$ JOB JNM=jobname,CLASS=class,USER=submitter name
* $$ LST CLASS=printer class
// JOB name
// EXEC DBSRFPR,SIZE=2048K
/*
/&
* $$ EOJ
```



# Appendix G: SQL Descriptor Area (SQLDA)

---

The SQL Descriptor Area (SQLDA) is used in the DESCRIBE statement. It can also be used in the EXECUTE, FETCH, OPEN, and PREPARE statements. Activities involving the SQLDA include building the SQLDA, updating the SQLDA, and using the SQLDA.

You build an SQLDA by coding an SQLDA and using a DESCRIBE or PREPARE statement to fill it. (When you use a PREPARE statement to fill an SQLDA, you must use the PREPARE's optional INTO clause.) For an example COBOL SQLDA and more information about the DESCRIBE and PREPARE statements, see [DESCRIBE](#) (see page 721) and [PREPARE](#) (see page 756).

After you have built an SQLDA in a DESCRIBE statement, you can update the SQLDATA entry (in the SQLDA) with a host variable address or update the SQLIND entry with an indicator variable address. You *must* update the SQLDATA entry with a host variable address before using the SQLDA in a FETCH statement.

You can retrieve values for parameter markers from the SQLDA by using OPEN or EXECUTE statements. A parameter marker is a question mark (?) that is used in place of a host variable in dynamic SQL statements. Rules for parameter markers can be found in PREPARE.

FETCH cursor-name can use the SQLDA to store the output of a select-statement, if that select-statement was built by the DECLARE CURSOR statement in which the corresponding cursor-name was defined.

## Determining Number of SQLVAR Entries to Use

Before the DESCRIBE is executed, the value of SQLN must be set to indicate how many occurrences of SQLVAR are provided in the SQLDA. To obtain the description of the columns of the result table of a prepared select-statement, the number of occurrences of SQLVAR must not be less than the number of columns.

When the USING BOTH clause is specified in the DESCRIBE statement, and SQLN is less than twice the size of the SQLD field, SQLD is set to twice the number of columns. When USING BOTH is specified, and SQLN is greater than or equal to twice the size of the SQLD field, SQLD is set to the number of columns.

The following technique is suggested for determining the number of SQLVAR entries to use:

1. Execute a DESCRIBE or PREPARE statement. The SQLDA that is used must not have any occurrences of SQLVAR. If you use a PREPARE statement, it must use the INTO clause.
2. If the value that is returned in the SQLD field is:
  - Greater than zero, use that value to allocate an SQLDA with the necessary number of occurrences of SQLVAR, then use that SQLDA to execute a DESCRIBE statement.
  - Equal to zero, it means that the prepared statement was not a select-statement, and that means that the prepared statement has no result table to be described.

## SQLDA (DESCRIBE or PREPARE INTO Statements)

The SQLABC and SQLN fields are input fields, and must be set before executing the DESCRIBE or PREPARE INTO statements. The other fields are output fields, and are filled in by the DESCRIBE statement. The SQLDA values expected by and assigned by the DESCRIBE statement are as follows:

Field	Value Expected or Returned
SQLAID	SQLDA (set by CA Datacom/DB)
SQLABC	Length of the SQLDA, equal to $SQLN * 44 + 16$ (set by CA Datacom/DB) <b>Note:</b> If USING BOTH is used, upon return from the DESCRIBE, if SQLN was not large enough for both column names and the <i>labels</i> information, SQLD has been set to twice the number of columns, the correct value for SQLN.
SQLN	Total number of occurrences of SQLVAR provided in this SQLDA (set by user—see <a href="#">Determining Number of SQLVAR Entries to Use</a> (see page 869)).
SQLD	Number of columns described by occurrences of SQLVAR, or twice this number when USING BOTH is specified (set by CA Datacom/DB).

Field	Value Expected or Returned
SQLVAR	<p>If the value returned for SQLD is 0, or greater than the value of SQLN, no values are assigned to occurrences of SQLVAR.</p> <p>If the value returned for SQLD is <math>n</math>, where <math>n</math> is greater than 0 but less than or equal to the value returned for SQLN, values are assigned to the first <math>n</math> occurrences of SQLVAR so that the first occurrence of SQLVAR contains a description of the first column of the table, the second occurrence of SQLVAR contains a description of the second column of the table, and so on. The description of a column consists of the values assigned to SQLTYPE, SQLLEN, and SQLNAME.</p> <p>If USING BOTH is used, the first <math>n</math> occurrences of SQLVAR contain column names (where they exist). The second <math>n</math> occurrences contain column labels. Occurrence <math>n+1</math> contains the label of column 1; occurrence <math>n+2</math>, the label of column 2; and so on.</p>

## Data Type of the Column and Null Value

The following applies when the value expected or returned has a code showing the data type of the column and whether it can contain null values, as shown below.

Field	Field	Value	Data Type	Nulls
SQLVAR	SQLTYPE	384/385	date	no/yes
SQLVAR	SQLTYPE	388/389	time	no/yes
SQLVAR	SQLTYPE	392/393	timestamp	no/yes
SQLVAR	SQLTYPE	436/437	numeric (zoned decimal)	no/yes
SQLVAR	SQLTYPE	448/449	varying-length character string	no/yes
SQLVAR	SQLTYPE	452/453	fixed-length character string	no/yes
SQLVAR	SQLTYPE	456/457	long, varying-length character string	no/yes
SQLVAR	SQLTYPE	464/465	varying-length graphic string	no/yes
SQLVAR	SQLTYPE	468/469	fixed-length graphic string	no/yes
SQLVAR	SQLTYPE	472/473	long varying-length graphic string	no/yes

Field	Field	Value	Data Type	Nulls
SQLVAR	SQLTYPE	480/481	floating-point	no/yes
SQLVAR	SQLTYPE	484/485	decimal (packed decimal)	no/yes
SQLVAR	SQLTYPE	496/497	large integer	no/yes
SQLVAR	SQLTYPE	500/501	small integer	no/yes

**Note:** On DESCRIBE, the type codes 384/385, 388/389, and 392/393 denote date, time, and timestamp, respectively. However, host variables do not have date/time data types, so character string variables must be used to retrieve date/time values. Thus, when the SQLDA describes host variables, these type codes denote fixed-length character string variables.

## Data Type of the Column

The following applies when a value depends on the data type of the columns.

Field	Field	Data Type	Value
SQLVAR	SQLLEN	Any string	The length attribute of the column, that is, the maximum number of characters in a value
SQLVAR	SQLLEN	FLOAT	8
SQLVAR	SQLLEN	INTEGER	4
SQLVAR	SQLLEN	SMALLINT	2
SQLVAR	SQLLEN	DECIMAL(p,s)	p in byte 1 s in byte 2
SQLVAR	SQLLEN	NUMERIC(p,s)	p in byte 1 s in byte 2
SQLVAR	SQLLEN	DATE	10
SQLVAR	SQLLEN	TIME	8
SQLVAR	SQLLEN	TIMESTAMP	26



## SQLVAR and SQLTYPE

Before you can use the SQLDA in an EXECUTE, FETCH, or OPEN statement, the SQLDATA field must be loaded with the address of the host-variable that is to receive data (for that column), while the SQLIND field must be loaded (if applicable) with the address of an indicator variable.

Field	Field	Value Expected or Returned
SQLVAR	SQLDATA	A value of -1 indicates FOR BIT DATA.
SQLVAR	SQLIND	Reserved.
SQLVAR	SQLNAME-VARCHAR	A varying-length string (VARCHAR(30)) which contains the unqualified name of the column, or the label of the column, depending on the value of USING (NAMES, LABELS, ANY, or BOTH). Also, a string of length 0 if a column name is to be used but no column name exists (for example, the column of the result table is an expression).

## SQLDA (EXECUTE, FETCH, or OPEN Statement)

For the EXECUTE, FETCH, or OPEN statement, the SQLDA which describes the host variables must contain the following values:

Field	Value Expected or Returned
SQLAID	Not used
SQLABC	Length of the SQLDA, equal to $SQLN * 44 + 16$ (set by CA Datacom/DB)
SQLN	Total number of occurrences of SQLVAR provided in this SQLDA (set by user, see <a href="#">Determining Number of SQLVAR Entries to Use</a> (see page 869))
SQLD	Number of host variables described by occurrences of SQLVAR

**Note:** The fields of each occurrence of SQLVAR must contain the following values.

Field	Field	Value	Data Type	Nulls
SQLVAR	SQLTYPE	384/385	fixed-length character string	no/yes

Field	Field	Value	Data Type	Nulls
SQLVAR	SQLTYPE	388/389	fixed-length character string	no/yes
SQLVAR	SQLTYPE	392/393	fixed-length character string	no/yes
SQLVAR	SQLTYPE	436/437	numeric (zoned decimal)	no/yes
SQLVAR	SQLTYPE	448/449	varying-length character string	no/yes
SQLVAR	SQLTYPE	452/453	fixed-length character string	no/yes
SQLVAR	SQLTYPE	456/457	long, varying-length character string	no/yes
SQLVAR	SQLTYPE	464/465	varying-length graphic string	no/yes
SQLVAR	SQLTYPE	468/469	fixed-length graphic string	no/yes
SQLVAR	SQLTYPE	472/473	long varying-length graphic string	no/yes
SQLVAR	SQLTYPE	480/481	floating-point	no/yes
SQLVAR	SQLTYPE	484/485	decimal (packed decimal)	no/yes
SQLVAR	SQLTYPE	496/497	large integer	no/yes
SQLVAR	SQLTYPE	500/501	small integer	no/yes
SQLVAR	SQLTYPE	504/505	COBOL display sign leading separate	no/yes

Defines the external length of a value from the host variable, as shown in the following table.

Field	Field	Data Type	Value
SQLVAR	SQLLEN	VARCHAR	Length attribute in bytes
SQLVAR	SQLLEN	GRAPHIC	Precision (number of double-byte characters)

Field	Field	Data Type	Value
SQLVAR	SQLLEN	VARGRAPHIC	Precision (max number of double-byte characters)
SQLVAR	SQLLEN	CHARACTER	Length attribute in bytes
SQLVAR	SQLLEN	FLOAT	8 (bytes)
SQLVAR	SQLLEN	INTEGER	4 (bytes)
SQLVAR	SQLLEN	SMALLINT	2 (bytes)
SQLVAR	SQLLEN	DECIMAL	Precision in byte 1, scale in byte 2
SQLVAR	SQLLEN	NUMERIC	Precision in byte 1, scale in byte 2
SQLVAR	SQLLEN	DATE	10 (bytes)
SQLVAR	SQLLEN	TIME	8 (bytes)
SQLVAR	SQLLEN	TIMESTAMP	26 (bytes)

## SQLVAR and VS/COBOL

Before you can use the SQLDA in an EXECUTE, FETCH, or OPEN statement, the SQLDATA field must be loaded with the address of the host variable that is to receive data (for that column), while the SQLIND field must be loaded (if applicable) with the address of an indicator variable.

In VS/COBOL II a "pointer" data type field can be associated with each host variable. The value in these pointer fields can be moved to the SQLDATA field to load the address of the host variable. In VS/COBOL, an Assembler subroutine must be used to determine the address of the host variable (see the example Assembler subroutine in the sample dynamic SQL program on the installation tape).

Field	Field	Value Expected or Returned
SQLVAR	SQLDATA	Contains the address of the host variable.
SQLVAR	SQLIND	If there is an associated indicator variable, SQLIND contains its address. If there is not an associated indicator variable, SQLIND is not used.
SQLVAR	SQLNAME	Not used