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

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

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

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

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

The manufacturer of this Documentation is CA.

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

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

This document references the following CA Technologies products:

- CA IDMS™/DB
- CA IDMS™ Presspack
- CA IDMS™ SQL
- CA IDMS™ Server

Contact CA Technologies

Contact CA Support

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

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

Providing Feedback About Product Documentation

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

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

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

- **CA IDMS Scalar Functions** (see page 124)—Updated the syntax diagram for the TRIM function.
- **Third-Party Acknowledgement** (see page 853)—Updated the copyright year.

The following documentation updates were made for the 18.5.00 release of this documentation:

- **CA ADS, COBOL, PL/I Data Types** (see page 849)—The information in this new appendix was previously available in the SQL Quick Reference Guide.
- **SQL Communication Area** (see page 675)—The following new subsections were added to this chapter from the SQL Quick Reference Guide:
  - **COBOL/CA ADS SQLCA** (see page 683)
  - **PL/I SQLCA** (see page 684)
  - **SQLCODE and SQLCNRP Values** (see page 681)
# Contents

## Chapter 1: Coding Considerations

Definitions ......................................................................................................................... 15
Using SQL Statements ................................................................................................. 16
SQL Comments ........................................................................................................... 18
Syntax Diagram Conventions .................................................................................. 19

## Chapter 2: Identifiers

About Identifiers ......................................................................................................... 24
Expansion of Authorization-identifier ........................................................................ 27
Expansion of Procedure-reference ............................................................................. 28
Expansion of Table-procedure-reference .................................................................. 32
Expansion of Table-name ......................................................................................... 35
Expansion of Table-reference ..................................................................................... 37
Expansion of Joined-table ......................................................................................... 39
Identifying Entities in Schemas ................................................................................. 42
Expansion of Cursor-name ......................................................................................... 45
Expansion of Statement-name ..................................................................................... 48

## Chapter 3: Data Types and Null Values

Data Types ..................................................................................................................... 53
Expansion of Data-type ............................................................................................. 55
Representation of Date/Time Values .......................................................................... 63
Comparison, Assignment, Arithmetic, and Concatenation Operations .................. 66
Null Values ................................................................................................................... 73

## Chapter 4: Values and Value Expressions

Literals ........................................................................................................................... 75
Expansion of Literal ................................................................................................... 75
Host Variables ............................................................................................................. 77
Expansion of Host-variable ....................................................................................... 79
Local Variables .......................................................................................................... 81
Expansion of Local-variable ...................................................................................... 81
Routine Parameters .................................................................................................... 84
Expansion of Routine-parameter ............................................................................... 85
Dynamic Parameters ................................................................................................... 87
Chapter 8: Statements

Statement Categories ........................................................................................................ 251
ALLOCATE CURSOR ........................................................................................................ 258
ALTER ACCESS MODULE ............................................................................................... 261
ALTER CATALOG ............................................................................................................. 269
ALTER CONSTRAINT ........................................................................................................ 270
ALTER FUNCTION ............................................................................................................ 272
ALTER INDEX .................................................................................................................. 277
ALTER PROCEDURE ......................................................................................................... 279
ALTER SCHEMA ............................................................................................................... 285
ALTER TABLE .................................................................................................................. 290
ALTER TABLE PROCEDURE ............................................................................................. 300
BEGIN DECLARE SECTION .............................................................................................. 305
CALL ............................................................................................................................... 306
CLOSE .............................................................................................................................. 313
COMMIT .......................................................................................................................... 315
CONNECT ....................................................................................................................... 317
CREATE ACCESS MODULE ............................................................................................. 320
CREATE CALC ................................................................................................................. 333
CREATE CONSTRAINT ..................................................................................................... 334
CREATE FUNCTION ......................................................................................................... 341
CREATE INDEX ............................................................................................................... 352
CREATE KEY .................................................................................................................. 357
CREATE PROCEDURE ..................................................................................................... 361
CREATE SCHEMA ........................................................................................................... 373
CREATE TABLE ............................................................................................................... 378
CREATE TABLE PROCEDURE ......................................................................................... 387
CREATE TEMPORARY TABLE .......................................................................................... 392
CREATE VIEW ................................................................................................................ 394
DEALLOCATE PREPARE ................................................................................................. 401
DECLARE CURSOR .......................................................................................................... 402
DECLARE EXTERNAL CURSOR ......................................................................................... 406
DELETE ........................................................................................................................... 408
DESCRIBE ....................................................................................................................... 413
DESCRIBE CURSOR ....................................................................................................... 416
DROP ACCESS MODULE ............................................................................................... 418
DROP CALC ..................................................................................................................... 420
DROP CONSTRAINT ........................................................................................................ 421
DROP FUNCTION ............................................................................................................ 423
Chapter 9: Control Statements

Overview .................................................................................................................................. 559
SQL Control Statements ........................................................................................................... 560
## Appendix F: Index Calculations 747
INDEX BLOCK CONTAINS ........................................................................................................... 748
DISPLACEMENT ......................................................................................................................... 749

## Appendix G: Sample COBOL Table Procedure 751
Sample Table Procedure Definition ......................................................................................... 751
Sample Table Procedure Program ........................................................................................... 752

## Appendix H: DISPLAY and PUNCH Syntax 769
DISPLAY and PUNCH Syntax ..................................................................................................... 769
DISPLAY and PUNCH Operations ............................................................................................. 770
DISPLAY/PUNCH ALL Statement ............................................................................................. 770
DISPLAY/PUNCH ACCESS MODULE ....................................................................................... 781
DISPLAY/PUNCH CALC KEY ..................................................................................................... 784
DISPLAY/PUNCH CONSTRAINT ................................................................................................. 786
DISPLAY/PUNCH FUNCTION ...................................................................................................... 788
DISPLAY/PUNCH INDEX ........................................................................................................... 791
DISPLAY/PUNCH KEY ............................................................................................................... 793
DISPLAY/PUNCH PROCEDURE ................................................................................................. 794
DISPLAY/PUNCH SCHEMA ....................................................................................................... 796
DISPLAY/PUNCH TABLE ........................................................................................................... 800
DISPLAY/PUNCH TABLE PROCEDURE .................................................................................... 805
DISPLAY/PUNCH VIEW .............................................................................................................. 808

## Appendix I: Sample COBOL Procedure 813
Sample Procedure Definition .................................................................................................... 813
Sample Procedure Program ...................................................................................................... 814
Sample of Procedure Invocation ............................................................................................... 816

## Appendix J: CA IDMS Scalar Functions 817
Overview ..................................................................................................................................... 817
Functions .................................................................................................................................... 817

## Appendix K: Sample COBOL Function 823
Sample Function Definition ....................................................................................................... 823
Sample Function Program ......................................................................................................... 824
Appendix L: Sample CA ADS Procedure

Appendix M: Sample CA ADS Function

Appendix N: SQL Cache Tables

Appendix O: Enhancing the Presentation of Access Strategy Information

Appendix P: SQL Reserved Words

Appendix Q: CA ADS, COBOL, PL/I Data Types

Appendix R: Third-Party Acknowledgment

Index
Chapter 1: Coding Considerations

This section contains the following topics:

Definitions (see page 15)
Using SQL Statements (see page 16)
SQL Comments (see page 18)
Syntax Diagram Conventions (see page 19)

Definitions

Most of the definitions and entities used in this guide are intuitively understood and generally used. With the introduction of SQL procedural language support it became necessary to more formally define a number of SQL routine-like objects. The following definitions are based on the definitions used in the SQL standard.

SQL-invoked routine

Specifies a routine that is allowed to be invoked only from within SQL. An SQL-invoked routine can be defined in the SQL catalog as a procedure, function, or table procedure.

SQL-invoked procedure

Specifies an SQL-invoked routine defined as a procedure in the SQL catalog.

SQL-invoked function

Specifies an SQL-invoked routine defined as a function in the SQL catalog.

SQL routine

Specifies an SQL-invoked routine whose language attribute is SQL. Because table procedures can not be written in the SQL language, an SQL routine is necessarily defined as a procedure or a function.

SQL procedure

Specifies an SQL routine defined in the SQL catalog as a procedure with language attribute SQL.

SQL function

Specifies an SQL routine defined in the SQL catalog as a function with language attribute SQL.
Using SQL Statements

You can submit SQL statements to CA IDMS by:

- Using the CA IDMS online command facility (that is, interactively)
- Using the CA IDMS batch command facility
- Embedding the statements in an application program (that is, programmatically)
- Using tools and facilities, (such as, CA IDMS Visual DBA, CA Visual Express) that submit SQL statements through CA IDMS Server

The same syntax applies no matter how you submit the statements. However, there are some statements that are only programmatic. Chapter 9, "Statements" indicates those statements that you submit only in SQL that is embedded in a program.

Statement Components

Keywords, Values, and Separators

SQL statements consist of:

- **Keywords** that:
  - Identify the action requested by the statement (for example, CREATE or SELECT)
  - Specify the type of entity (for example, TABLE or INDEX) that is the object of the requested action
  - Place qualifications on the requested action, either by themselves (for example, NOT NULL or DISTINCT) or in conjunction with user-supplied values (for example, ORDER BY EMPLOYEE_LNAME)

- **User-supplied values** that:
  - Identify specific occurrences of entities (for example, the EMPLOYEE table or user EKI)
  - Specify data values (for example, 983 or 'Boston')

- **Separators** that separate keywords and user-supplied values from one another. A separator can be a space, a comment, a new-line character, or the end of the line.

Where Separators Are Not Required

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>:</td>
<td>Colon</td>
</tr>
</tbody>
</table>
Comma

Equal sign

Not equal sign

Greater than or equal to sign

Greater than sign

Not greater than sign

Left and right parentheses

Less than or equal to sign

Less than sign

Not less than sign

Minus sign

Not equal sign

Period

Plus sign

Semicolon

Slash

Concatenation sign

**Uppercase and Lowercase**

You can use both uppercase and lowercase to enter keywords and user-supplied values in SQL statements. CA IDMS converts lowercase letters to uppercase in keywords and in user-supplied values that are not enclosed in quotation marks.

**Delimiting and Continuing Statements**

**Statement Delimiter for the Command Facility**

When you use the command facility to submit SQL statements, you must terminate each statement with a command delimiter, which is by default a semicolon (;). You can enter the command delimiter either on the same line as the rest of the statement or on a separate line. For example, the following two statements are equivalent:

```
select * from employee;
```

```
select * from employee
```


Continuing Statements

You can code SQL statements on one or more lines. No special character is required to indicate that a statement continues on the next line.

Embedded SQL Delimiters

When you embed SQL statements in an application program or a CA ADS process module, you must delimit each statement both at the beginning and at the end. The requirements for delimiting embedded SQL statements vary according to the program language.

Note: For more information about delimiting embedded SQL statements, see the CA IDMS SQL Programming Guide.

SQL Comments

How to Embed SQL Comments

You can embed an SQL comment within an SQL statement. SQL comments may be used in both interactive and embedded SQL statements.

An SQL comment:
- Begins with two consecutive hyphens (--) 
- Consists of any combination of numbers, letters, spaces, and other characters
- Ends at the end of the line

Bracketed comment:
- A bracketed comment starts with the bracket introducer string '/*' and ends with the bracket terminator string '*/'.
- Bracketed comments can span multiple lines.
- The bracket introducer and terminator strings can not split over two lines.
- Can be used whenever a separator or space is allowed.
- Bracketed comments are only allowed in the routine body of an SQL-routine. They are not recognized by the command facility outside this context.
Note: When defining an SQL routine using the command facility tools OCF, IDMSBCF, or using an OCF console in CA IDMS Visual DBA, the comment introducer ‘/*’ must not be placed in column 1, because ‘/’ is interpreted as an end of file on input by the command facility.

Sample SQL Comments

The following example shows SQL comments with an embedded SQL statement:

```sql
select
  emp_id, emp_lname, dept_id  — Columns to be selected
from employee               — Tables containing the data
where dept_id = 1234;       — Selection criterion
```

Syntax Diagram Conventions

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

- **UPPERCASE OR SPECIAL CHARACTERS**
  - Represents a required keyword, partial keyword, character, or symbol that must be entered completely as shown.

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

- **italicized lowercase**
  - Represents a value that you supply.

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

- **◄─**
  - Points to the default in a list of choices.

- **►►────────────────────**
  - Indicates the beginning of a complete piece of syntax.

- **_______________**
  - Indicates the end of a complete piece of syntax.

- **◄─────────────────────**
  - Indicates that the syntax continues on the next line.

- **_________**
  - Indicates the end of a complete piece of syntax.
Indicates that the syntax continues on this line.

Indicates that the parameter continues on the next line.

Indicates that a parameter continues on this line.

Indicates a required parameter.

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

Indicates an optional parameter.

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

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

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

The following sample explains how the notation conventions are used:

[Diagram of Syntax Diagram conventions]

- Beginning of the syntax
- Required parameter
- Optional portion of parameter
- User-supplied value
- Syntax continues on the next line

- Syntax continues on this line
- Comma required between repetition
- Repetition allowed

- Optional keyword
- Select one or none
- Default
- Portion of syntax expanded elsewhere
- End of the syntax

- Keyword
- Variable
Chapter 2: Identifiers

This section contains the following topics:

- About Identifiers (see page 24)
- Expansion of Authorization-identifier (see page 27)
- Expansion of Procedure-reference (see page 28)
- Expansion of Table-procedure-reference (see page 32)
- Expansion of Table-name (see page 35)
- Expansion of Table-reference (see page 37)
- Expansion of Joined-table (see page 39)
- Identifying Entities in Schemas (see page 42)
- Expansion of Cursor-name (see page 45)
- Expansion of Statement-name (see page 48)
About Identifiers

Identifiers are the smallest lexical units used in entity names.

The following entities referenced in SQL statements have identifiers:
- Access modules
- Areas
- Columns
- Constraints
- Cursors
- Groups
- Indexes
- Keys
- Procedures
- RCMs (SQL statement modules)
- Referential constraints
- Schemas
- Segments
- Statement names
- Tables
- Table procedures
- User-defined functions
- Users
- Views

Qualifying Identifiers

Identifiers for some entities can be qualified by other identifiers. For example, table identifiers can be qualified by schema names.

Sometimes an identifier by itself does not uniquely identify an entity. For instance, a SELECT statement may include two columns with the same identifier, each from a different table. To uniquely identify each of these columns, you must qualify each column identifier with the associated table name or alias.
To qualify an identifier, specify the qualifier first, followed by a period (.), followed by the identifier you are qualifying. The qualified identifier in the following example identifies the EMPLOYEE table associated with the DEMOEMPL schema:

demoEMPL.employee

**Authorization-identifier and table-name**

**Authorization-identifier** and **table-name** are syntactic elements representing identifiers that occur in multiple SQL statements. For expanded syntax for these elements, see Expansion of Authorization-identifier and Expansion of Table-name.

### Forming Identifiers

**Valid characters**

An identifier consists of a combination of:

- Letters (A through Z and a through z)
- Digits (0 through 9)
- At sign (@)
- Dollar sign ($)
- Pound sign (#)
- Underscore (_)

The first character of an identifier must be a letter, @, $, or #.

**Maximum length**

Identifiers for all entities except columns, access modules, segments, RCMs, and external names of SQL-invoked routines can be as many as 18-characters long.

Identifiers for access modules, segments, RCMs, and external names of SQL-invoked routines can be as many as eight-characters long.

An identifier for a column can be as many as 32-characters long.
Delimited Identifiers

**Why delimit identifiers**

You can delimit an identifier in double quotation marks to:

- **Allow the use of special characters and blanks.** An identifier enclosed in quotation marks can consist of any combination of characters. For example, the following is a valid identifier:

  ```sql
  "&ATM*F(0517). MA"
  ```

  To include a double quotation mark as part of the identifier itself, use two consecutive double quotation marks. For example:

  ```sql
  "M1K" "L96ZZ".
  ```

- **Make case significant.** When you enclose an identifier in quotation marks, CA IDMS does not convert lowercase letters to uppercase.

  Lowercase letters in quotation marks are not equal to uppercase letters or to lowercase letters that are not in quotation marks. In the example below, the identifiers on the left all identify the same table; the identifier on the right identifies a different table:

  ```sql
  employee               "employee"
  EMPLOYEE              "EMPLOYEE"
  ```

**Placement of quotation marks**

If one or more parts of a qualified identifier require quotation marks, place the quotation marks only around the individual parts. Do not include two identifiers in one set of quotation marks. For example, both parts of the following qualified identifier require quotation marks:

```sql
"temp-tab-1"."Commission to Date"
```

When you calculate the length of an identifier, do not include delimiting quotation marks.

**Avoiding Keywords as Identifiers**

**Eliminating ambiguity**

You should avoid issuing an SQL statement which specifies an identifier that matches a keyword in the syntax for the statement. This eliminates potential ambiguity that could cause CA IDMS to read the statement in a way that is not meant.

If you must use a keyword as an identifier, delimit the identifier with double quotation marks as described in Delimited Identifiers.
Recognizing keywords

The syntax diagrams in this guide present keywords in roman (that is, nonitalicized) type.

For example, in the following syntax for CREATE VIEW, the keywords are CREATE VIEW, AS, and WITH CHECK OPTION:

```
CREATE VIEW [schema-name.] view-identifier

( [view-column-name] )

AS query-specification

order-by-specification

WITH CHECK OPTION
```

Delimited example identifier

If it were necessary to create a view called AS, you should identify the view as follows:

create view "AS" (col1, col2, col3) as (select...

Expansion of Authorization-identifier

The expanded parameters of authorization-identifier represent user identifiers or group identifiers in an SQL authorization statement.

Syntax

```
Expansion of authorization-identifier

user-identifier

group-identifier
```
Parameters

user-identifier

Identifies a user defined to the security system.

group-identifier

Identifies a group defined to the security system.

Examples

Authorizing a User to Update a Table

In the following GRANT statement, the authorization identifier is the user identifier RES:

grant update
    on table employee
    to res;

Revoking Execute Privileges from a Group

In the following REVOKE statement, the authorization identifier is the group identifier ACCT_GRP_1:

revoke execute
    on access module am88pr08
    from acct_grp_1;

Expansion of Procedure-reference

The expanded parameters of procedure-reference represent qualified or unqualified procedure identifiers together with an optional set of parameter values.

If an SQL CALL or an SQL SELECT statement that is embedded in an application program or SQL routine contains the procedure reference, then the procedure reference also identifies the target host variables, local variables, or routine parameters into which the output parameter values return.
Syntax

Expansion of procedure-reference

\[
\[
\text{schema-name.} \quad \text{procedure-identifier} \\
( \text{parameter-specification} ) \\
\]

Expansion of parameter-specification

\[
\text{parameter-name} \quad = \quad \text{value-expression} \\
\]

Parameters

\text{schema-name}

Specifies the schema with which the procedure identified by \text{procedure-identifier} is associated.

\textbf{Note}: For more information about using a schema name to qualify a procedure, see Identifying Entities in Schemas.

\text{procedure-identifier}

Identifies a procedure defined in the dictionary.

\text{parameter-specification}

Specifies a value assigned to a parameter of a procedure. If an SQL CALL or an SQL SELECT statement that is embedded in an application program contains the procedure and the value-expression is a host-variable, then the output value of the parameter returns into the specified host-variable. If the SQL CALL or SQL SELECT statement is embedded in an SQL routine and the value-expression is a local variable or a routine parameter then the output value of the parameter returns into the specified local variable or routine parameter.

You can use both the positional (with NO \text{parameter-name}) and the non-positional (with \text{parameter-name}) forms of parameter specification in a single procedure reference. If you use a non-positional parameter specification, then all remaining parameter specifications in the parameter list MUST be non-positional. Positional parameter specifications are assumed to correspond to the declared parameters of a procedure in the sequence of their declaration.

\text{parameter-name}

Specifies the name of a parameter associated with the procedure.

\text{value-expression}

Specifies the input value to assign to the parameter. In addition, any host-variable, local variable, or routine parameter specified as value-expression receives the output value of the parameter returned by the invoked procedure. See Expansion of Value-expression for more information.
Usage

Referencing Procedures

You can code references to SQL procedures in an SQL CALL statement.

During SQL CALL processing, CA IDMS issues a call to the corresponding routines. The output parameter values return as a result set.

You can also reference a procedure in the FROM clause of a query-specification or SELECT statement, in the same manner as references to SQL tables, views, and table procedures.

If you reference a procedure in a FROM clause, then the parameters of the procedure act as columns in an SQL table or view. You can reference them in SELECT list expressions and WHERE clauses. A procedure returns exactly one row of output or no output.

Assigning Parameter Values with the WHERE Clause

You can use the WHERE clause as an alternative method for assigning values to parameters of procedures. An expression of the form parameter name = value specification coded in the WHERE clause is considered to be equivalent to a parameter assignment using procedure reference syntax. This allows you to code procedure references without a parenthesized parameter list, just like standard table, view or table procedures references.

This method is useful particularly if you are coding SQL statements in generic SQL environments, such as CA Visual Express, which do not support the SQL CALL statement and the specification of parameters in the procedure reference.

When you use the WHERE clause to assign parameter values, you must meet the following conditions in order to assign the parameter a value:

- It must appear in an "=" comparison, not, for example, with >, <, >=, or <=.
- You can combine the "=" comparison in which the parameter appears only with other factors in the WHERE clause using an AND operator. Use of an OR operator or preceding the "=" comparison with the NOT keyword means that no value is assigned to the parameter.

Note: For more information about assignment of values to procedure parameters, see Procedure Parameters.
Examples

Qualified Procedure Reference through the CALL Statement

In the following CALL statement, the procedure reference is qualified and one parameter value is supplied as a positional parameter for the first parameter of the procedure get_bonus:

call emp.get_bonus (127);

Procedure Reference with Keyword Parameter Values

In the following CALL statement, a value is supplied for the EMP_ID parameter using keyword notation:

call get_bonus (emp_id=127);

Procedure Reference through the SELECT Statement

In the following SELECT statement, a value is supplied for the first parameter associated with the GET_BONUS procedure:

select * from get_bonus (7);

Procedure Reference through the SELECT Statement with Parameter Values Specified in the WHERE Clause;

In the following SELECT statement, parameter values are supplied through the WHERE clause. This example is identical to the example above that uses keyword notation:

select * from get_bonus
where emp_id=127;

Note: For more information about defining procedures, see CREATE PROCEDURE and Defining and Using Procedures.

More information

Procedure Parameters (see page 631)

More information:

CREATE PROCEDURE (see page 361)
Defining and Using Procedures (see page 629)
Expansion of Table-procedure-reference

More information

CREATE TABLE PROCEDURE (see page 387)
Defining and Using Table Procedures (see page 617)

Expansion of Table-procedure-reference

The expanded parameters of table-procedure-reference represent qualified or unqualified table procedure identifiers together with an optional set of parameter values.

Syntax

```
Expansion of table-procedure-reference

schema-name. table-procedure-identifier

( parameter-specification )

Expansion of parameter-specification

parameter-name = value-expression
```

Parameters

`schema-name`

Specifies the schema with which the table procedure identified by `table-procedure-identifier` is associated.

**Note:** For more information about using a schema name to qualify a table procedure, see Identifying Entities in Schemas.

`table-procedure-identifier`

Identifies a table procedure defined in the dictionary.
**Usage**

**Referencing Table Procedures**

You can code references to SQL table procedures in SQL SELECT, INSERT, UPDATE, and DELETE statements in the same manner as references to SQL tables and views. The parameters of such table procedures act as columns in an SQL table or view. You can reference them in SELECT list expressions, WHERE clauses, UPDATE statement SET clauses, and the column list of the INSERT statement. You can also reference table procedures in the SQL CALL statement. The output parameter values return as a result set.

During SQL DML processing, CA IDMS issues calls to the corresponding external routines at the same time at which it would perform database access to satisfy standard table references. This permits the simulation of SQL DML activity on external data storage structures (for example, non-SQL-defined CA IDMS databases or VSAM file systems) managed by the table procedures.

**Assigning Parameter Values with the WHERE Clause**

An alternative method for assigning values to parameters of table procedures is through the WHERE clause. An expression of the form `parameter name = value specification` coded in the WHERE clause is considered to be equivalent to a parameter assignment using table procedure reference syntax. This allows table procedure references to be coded without a parenthesized parameter list, just like standard table or view references.

This method is useful particularly if you are coding SQL statements in generic SQL environments, such as CA Visual Express, which do not support CA IDMS SQL extensions, such as table procedures.
When you use the WHERE clause to assign parameter values, the following conditions must be met for the parameter to be assigned a value:

- It must appear in an "=" comparison, not, for example, with >, <, >=, or <=.
- The "=" comparison in which the parameter appears can be combined only with other factors in the WHERE clause using an AND operator. Use of an OR operator or preceding the "=" comparison with the NOT keyword means that no value is assigned to the parameter.

**Note:** For more information about assignment of values to table procedure parameters, see Table Procedure Parameters.

### Examples

**Qualified Table Procedure Reference**

In the following SELECT statement, the table procedure reference is qualified:

```
select * from emp.org;
```

**Table Procedure Reference with Keyword Parameter Values**

In the following SELECT statement, values are supplied for the EMP_ID and MGR_ID parameters using keyword notation:

```
select * from org (emp_id=127, mgr_id=7);
```

**Table Procedure Reference with Positional Parameter Values**

In the following SELECT statement, a value is supplied for the first parameter associated with the ORG table procedure:

```
select * from org (7);
```

**Table Procedure Reference with Parameter Values Specified in the WHERE Clause**

In the following SELECT statement, parameter values are supplied through the WHERE clause. This example is identical to the example above that uses keyword notation.

```
select * from org .
    where emp_id=127 and mgr_id=7;
```

**Note:** For more information about defining table procedures, see CREATE TABLE PROCEDURE and Defining and Using Table Procedures.
More information:

Table Procedure Parameters (see page 619)

Expansion of Table-name

The expanded parameters of table-name represent qualified or unqualified tables, views, procedures, or table procedure identifiers in an SQL statement.

Syntax

```
Expansion of table-name

        schema-name. [table-identifier]
          view-identifier
            procedure-identifier
              table-procedure-identifier
```

Parameters

**schema-name**

Specifies the schema with which the table, view, procedure, or table procedure identified by table-identifier, view-identifier, procedure-identifier or table-procedure-identifier is associated.

**Note:** For more information about using a schema name to qualify a table, view, procedure, or table procedure identifier, see Identifying Entities in Schemas. (see page 42)

**table-identifier**

Identifies either a base table defined in the dictionary or a temporary table defined during the current transaction.

**view-identifier**

Identifies a view defined in the dictionary.

**procedure-identifier**

Identifies a procedure defined in the dictionary.

**table-procedure-identifier**

Identifies a table procedure defined in the dictionary.
Expansion of Table-name

Examples

A Qualified Table Identifier

In the following INSERT statement, the table name is a qualified table identifier:

```sql
insert into demoempl.employee values (1,'John', 'Smith');
```

An Unqualified Table Identifier

In the following INSERT statement, the table name is an unqualified table identifier. If no temporary table named EMPLOYEE has been defined during the current transaction, CA IDMS assumes the table is qualified with the current schema in effect for the SQL session.

```sql
insert into employee values (1,'John', 'Smith');
```

More Information

- For more information about defining base tables, see CREATE TABLE, ALTER TABLE and DROP TABLE.
- For more information about defining temporary tables, see CREATE TEMPORARY TABLE.
- For more information about defining procedures, see CREATE PROCEDURE and Defining and Using Procedures.
- For more information about defining table procedures, see CREATE TABLE PROCEDURE and Defining and Using Table Procedures.
- For more information about defining views, see CREATE VIEW and DROP VIEW (see page 433).
- For more information about establishing a current schema, see SET SESSION.

More information:

- **ALTER TABLE** (see page 290)
- **CREATE PROCEDURE** (see page 361)
- **CREATE TABLE** (see page 378)
- **CREATE TEMPORARY TABLE** (see page 392)
- **CREATE VIEW** (see page 394)
- **Defining and Using Table Procedures** (see page 617)
- **DROP TABLE** (see page 430)
- **DROP VIEW** (see page 433)
- **SET SESSION** (see page 536)
Expansion of Table-reference

The expanded parameters of table-reference represent qualified or unqualified tables, view identifiers, joined tables, or a reference to a procedure or a table procedure in an SQL statement.

Syntax

```
Expansion of table-reference
```

```
<schema-name> | <table-identifier>
| <view-identifier>
| <procedure-reference>
| <table-procedure-reference>
| <joined-table>
```

Parameters

**schema-name**

Specifies the schema with which the table or view identified by *table-identifier* or *view-identifier* is associated.

**Note:** For more information about using a schema name to qualify a table or view identifier, see Identifying Entities in Schemas. (see page 42)

**table-identifier**

Identifies either a base table defined in the dictionary or a temporary table defined during the current transaction.

**view-identifier**

Identifies a view defined in the dictionary.

**procedure-reference**

Identifies a procedure defined in the dictionary and optionally supplies parameter values to be passed to the procedure.

**Note:** For more information about the expansion of *procedure-reference*, see Expansion of Procedure-reference. (see page 28)

**table-procedure-reference**

Identifies a table procedure defined in the dictionary and optionally supplies parameter values to be passed to the table procedure.

**Note:** For more information about the expansion of *table-procedure-reference*, see Expansion of Table-procedure-reference (see page 32).

**joined-table**

Identifies a table that is derived from joining two specified tables.

**Note:** For more information about the expansion of *joined-table*, see Expansion of Joined-table. (see page 39)
Examples

A Qualified Table Identifier

In the following SELECT statement, the table reference is a qualified table identifier:

```sql
select * from demoempl.employee;
```

An Unqualified Table Identifier

In the following SELECT statement, the table reference is an unqualified table identifier. If no temporary table named EMPLOYEE has been defined during the current transaction, CA IDMS assumes the table is qualified with the current schema in effect for the SQL session.

```sql
select * from employee;
```

Table Procedure Reference with Keyword Parameter Values

In the following SELECT statement, the table reference is a table procedure reference where values are supplied for the EMP_ID and MGR_ID parameters using keyword notation:

```sql
select * from org (emp_id=127, mgr_id=7);
```

More Information

- For more information about defining base tables, see CREATE TABLE, ALTER TABLE and DROP TABLE.
- For more information about defining temporary tables, see CREATE TEMPORARY TABLE.
- For more information about defining procedures, see CREATE PROCEDURE and Defining and Using Procedures.
- For more information about defining table procedures, see CREATE TABLE PROCEDURE and Defining and Using Table Procedures.
- For more information about defining views, see CREATE VIEW and DROP VIEW.
- For more information about establishing a current schema, see SET SESSION.
Expansion of Joined-table

The expanded parameters of joined-table represent a table that is derived from joining two specified tables. A join operation on two tables is the result of the cross product of the two tables. A qualified join is followed by a filter operation. The cross or Cartesian product of two tables, left and right, is the result of extending each row of the left table with every row of the right table. The different types of join operations are specified through the following join types:

- CROSS
- UNION
- INNER
- LEFT OUTER
- RIGHT OUTER
- FULL OUTER

Syntax

Expansion of unqualified-joined-table

Expansion of qualified-joined-table

Expansion of join-condition

Chapter 2: Identifiers  39
Expansion of Joined-table

Parameters

unqualified-joined-table
  Specifies a joined-table where the join operation is a cross or union.

qualified-joined-table
  Specifies a joined-table where the join operation is an inner, left outer, right outer, or full outer.

table-reference
  Represents a table-like object. In a joined-table specification, a left and a right table-reference are required to define the left and right components of the join operation.

AS alias-l
  Defines a new name used to identify the left table-like object within the joined-table specification. Alias-l must be a 1-through 18-character name that follows the conventions for SQL identifiers.

AS alias-r
  Defines a new name used to identify the right table-like object within the joined-table specification. Alias-r must be a 1-through 18-character name that follows the conventions for SQL identifiers.

CROSS
  Specifies a cross join. A cross join is the cross product of the left and right table.

UNION
  Specifies a union join. A union join is equivalent to a full outer join where the join-condition always evaluates to false.

INNER
  Specifies an inner join. In an inner join, the cross product of the left and right table-like objects is made, and only the rows for which join-condition evaluates to true are kept in the result. This is the default.

LEFT/LEFT OUTER
  Specifies a left outer join. In a left outer join, the cross product of the left and right table-like objects is made, and the rows for which join-condition evaluates to true are kept. The result is extended with all the missing rows from the left table, and the values of the columns in the result row, derived from the right table, are set to NULL.
Expansion of Joined-table

Chapter 2: Identifiers

RIGHT/RIGHT OUTER

Specifies a right outer join. In a right outer join, the cross product of the left and right table-like objects is made, and the rows for which join-condition evaluates to true are kept. The result is extended with all the missing rows from the right table, and the values of the columns in the result row, derived from the left table, are set to NULL.

FULL/FULL OUTER

Specifies a full outer join. In a full outer join, the cross product of the left and right table-like objects is made, and the rows for which join-condition evaluates to true are kept. The result is extended with all the missing rows from the left table, and the values of the columns in the result row, derived from the right table, are set to NULL. The result is further extended with all the missing rows from the right table, and the values of the columns in the result row, derived from the left table, are set to NULL.

join-condition

Represents the truth condition for joining two table-like objects. Expanded syntax for join-condition appears immediately after the joined-table syntax. If join-condition contains a set-specification both the left and the right table-reference must specify base tables of a non-SQL-defined database that identify the owner and member of the non-SQL set.

Usage

- If a join type is not specified, INNER is assumed.
- Joined-tables can be nested. Evaluation is from left to right.
- It is advisable to use parenthesis when nesting joins.
- In a nested joined-table, only the join-condition of the inner most join can contain a set-specification because a set-specification requires that the left and right table-reference are base tables of a non-SQL-defined database.
- A query-expression that contains a joined-table is not updateable.

Examples

Selecting all Departments and Employees in Department

The following examples list all the departments and the employees of the department. The two statements give identical results.

```sql
select d.*, e.*
from DEMOEMPL.DEPARTMENT d left join DEMOEMPL.EMPLOYEE e
  on d.dept_id = e.dept_id
select d.*, e.*
from DEMOEMPL.EMPLOYEE e right join DEMOEMPL.DEPARTMENT d
  on d.dept_id = e.dept_id
```
Selecting all Depts./Empls. in Dept. with or without Position

The following examples show nesting of joined tables. The two statements give identical results.

```sql
select d.*, e.*, p.*
  from DEMOEMPL.DEPARTMENT d left join
       (DEMOEMPL.EMPLOYEE e left join DEMOEMPL.POSITION p
           on p.EMP_ID  = e.EMP_ID )
       on e.DEPT_ID = d.DEPT_ID;

select d.*, e.*, p.*
  from DEMOEMPL.DEPARTMENT d left join
       (DEMOEMPL.POSITION p right join DEMOEMPL.EMPLOYEE e
           on p.EMP_ID  = e.EMP_ID )
       on e.DEPT_ID = d.DEPT_ID;
```

**Note:** For more information about expansion of table-reference, see Expansion of Table-reference.

**More information:**

[Expansion of Table-reference](#) (see page 37)

## Identifying Entities in Schemas

Access modules, referential constraints, tables, views, table procedures, procedures, and user-defined functions are all associated with schemas. However, when you name one of these entities in an SQL statement, specification of the schema name is optional. The schema CA IDMS uses when processing the statement depends on the following:

- Type of entity
- Presence or absence of the schema name in the entity reference
- Statement where the reference to the entity occurs
- Method you use to submit the statement to CA IDMS (for example, through the online command facility or embedded in an application program)
Resolving References to Entities in Schemas

Interactive and dynamic SQL

When compiling a statement using interactive or dynamic SQL, or when using the EXPLAIN statement to determine the access strategy to be used for an SQL statement, CA IDMS resolves references to entities in schemas as follows:

- For each entity qualified by a schema name, CA IDMS uses the named schema.
- For each unqualified table, view or table procedure in a SELECT, INSERT, UPDATE, and DELETE statement, or one of them in an EXPLAIN statement:
  - CA IDMS looks for the definition of a temporary table created during the current database transaction whose name matches the specified identifier. If a match is found, CA IDMS assumes that the reference is to the temporary table.
  - If a matching temporary table name is not found, CA IDMS uses the current schema in effect for the SQL session.
- For all other unqualified references, CA IDMS uses the current schema in effect for the SQL session

Creating or altering an access module

When creating or altering an access module, the user can change the names of schemas in table, view, table procedure, procedure, and user-defined function references in SQL data manipulation statements. The user does this by specifying one or more schema mapping rules, each of which supplies a replacement for a schema name. This facility allows a single program (and its associated RCM) to reference one set of tables when it is included in one access module, and another set of tables when it is included in another access module with different schema mapping rules.
Identifying Entities in Schemas

When compiling an SQL statement during the creation or alteration of an access module, CA IDMS resolves schema names as follows:

1. For qualified references to schema entities in non-data manipulation statements, it uses the schema name specified.

2. For each qualified table, view, table procedure, procedure, or user-defined function reference in a data manipulation statement, CA IDMS uses the replacement schema name as specified in the schema mapping rules. If no replacement has been specified, it uses the schema name specified in the table, view, procedure, or user-defined function reference.

3. For unqualified references to schema entities in data description statements, it uses the schema name of the access module being created or altered.

4. For each unqualified table, view, table procedure, procedure, or user-defined function reference in a data manipulation, CA IDMS uses the replacement schema name for the NULL entry in the schema mapping rules. If no such entry is found, it uses the schema name of the access module being created or altered.

5. For all unqualified references to schema entities, CA IDMS uses the current schema associated with your SQL session.

After resolving the schema names according to the above rules, if CA IDMS cannot find the definition of a table, view, table procedure, procedure, or user-defined function referenced in a data manipulation statement, the statement remains uncompiled and a warning is issued. CA IDMS attempts to recompile the statement at runtime because the table-like object may have been created since compile time.

Automatic access module recreation

CA IDMS automatically recreates an access module at runtime if:

- CA IDMS encounters a statement that was not previously compiled
- The AUTO RECREATE option for the access module is ON and:
  - CA IDMS encounters a statement that refers to a table, view, table procedure, procedure, or user-defined function whose definition has changed since the access module was created or last altered
  - The application program has been recompiled since the access module was created or last altered
When it automatically recreates an access module, CA IDMS resolves references to schemas using the same rules as were used when the access module was created or last altered except in dealing with unqualified table, view, table procedure, procedure or user-defined function references in data manipulation and EXPLAIN statements:

- If the table, view, or table procedure reference is unqualified, CA IDMS first looks for the definition of a temporary table created within the current transaction whose name matches the specified table identifier. If one is found, the reference is assumed to be a reference to that temporary table.
- If no temporary table with the same name is found, then CA IDMS uses the schema mapping rules associated with the access module to resolve table, view, and table procedure references as described above in Creating or altering an access module.

**Note:** For more information about automatic access module re-creation, see CREATE ACCESS MODULE (see page 320) or the CA IDMS SQL Programming Guide.

### Expansion of Cursor-name

The expanded parameters of cursor-name represent a cursor.

#### Syntax

```
Expansion of cursor-name

static-cursor-name

extended-cursor-name
```

```
Expansion of static-cursor-name

cursor-name
```

```
Expansion of extended-cursor-name

LOCAL

GLOBAL

'cursor-name'

:host-variable

declaration-variable

typical-variable
```

#### Parameters

- **cursor-name**
  
  Specifies the name of the cursor as an identifier.

- **'cursor-name'**
  
  Specifies the name of the cursor as a literal whose value must conform to the rules for an identifier.

- **:host-variable**
  
  Specifies the name of the cursor as a host-variable whose value must conform to the rules for an identifier.
local-variable

Specifies the name of the cursor as a local-variable whose value must conform to the rules for an identifier.

routine-parameter

Specifies the name of the cursor as a routine-parameter whose value must conform to the rules for an identifier.

LOCAL/GLOBAL

Specifies the scope of the associated cursor name:

- LOCAL indicates that the cursor can be referenced only from within the program where it is defined.
- GLOBAL indicates that the cursor can be referenced from any program executing within the same SQL transaction.

Default: LOCAL

Usage

Static Versus Extended Cursor Names

A static cursor name is one coded as a simple identifier. The following DECLARE CURSOR statement assigns the static name "cursor1" to the cursor being defined:

DECLARE cursor1 CURSOR FOR select1

Cursors defined by a DECLARE CURSOR statement always have static names. Such cursors may either be dynamic or static, depending on whether the DECLARE CURSOR statement references a dynamically prepared SQL statement, as in the example above, or directly includes a cursor-specification.

An extended cursor name is one coded either as a literal, a host variable, a local-variable, or a routine-parameter. The following ALLOCATE CURSOR statement assigns the extended name "cursor1" to the cursor being defined:

MOVE 'cursor1' to cursor
ALLOCATE :cursor CURSOR FOR :select

Cursors created by an ALLOCATE CURSOR statement always have extended names and are always dynamic.

If a cursor is defined using a static name, it must be referenced using a static name. If it is defined using an extended name, it must be referenced using an extended name that has the same scope option as specified on the definition.
An exception to this rule occurs when identifying a cursor within a dynamically prepared UPDATE or DELETE statement.

**Note:** For more information about the UPDATE and DELETE statements, see UPDATE (see page 546) and DELETE (see page 408).

### Uniqueness of Cursor Names

Static and extended cursor names do not have to be unique with respect to each other. If two cursors are assigned the same value for a name, they are considered two separate cursors provided that either:
- One of the names is static while the other is extended
- Both of the names are extended, but they have different scopes, as indicated by their LOCAL/GLOBAL parameter.

#### Example

**Example of Cursor-name**

The following DECLARE CURSOR statement defines a dynamic cursor using a static cursor name of C1. It is referenced within the subsequent OPEN statement:

```sql
EXEC SQL
  DECLARE C1 CURSOR FOR S1
END-EXEC
EXEC SQL
  OPEN C1
END-EXEC
```

**Example of Extended-cursor-name**

The following ALLOCATE CURSOR statement defines a local cursor using an extended cursor name of C1. It is then referenced in the subsequent OPEN statement:

```sql
EXEC SQL
  ALLOCATE 'C1' CURSOR FOR 'S1'
END-EXEC
EXEC SQL
  OPEN 'C1'
END-EXEC
```
Note: Even though C1 is used as the cursor name in both of the above examples, two separate cursors are created: one with a static name of C1 and one with an extended name of C1.

Global Extended-cursor-name

The following ALLOCATE CURSOR statement defines a global cursor using an extended cursor name whose value is not known until runtime. In this case, the value 'C2' is moved to the host variable before the statement is executed and will be the name of the cursor created:

```sql
move 'C2' to :cname
EXEC SQL
    ALLOCATE GLOBAL :CNAME CURSOR FOR :SNAME
END-EXEC
```

Since this is a global cursor, it can be referenced in a different program than the one where the ALLOCATE CURSOR statement appears. For example, the following OPEN statement might be contained in a different program:

```sql
EXEC SQL
    OPEN GLOBAL 'C2'
END-EXEC
```

Note: It does not matter that in one case the name of the cursor is supplied through a host-variable and in the other it is specified as a literal. They both refer to the global cursor C2.

More information:

- Defining and Using Functions (see page 639)
- Identifying Entities in Schemas (see page 42)

Expansion of Statement-name

The expanded parameters of statement-name represent a dynamically-prepared statement.

Syntax

```
Expansion of statement-name

static-statement-name

extended-statement-name

Expansion of static-statement-name

statement-name
```
Expansion of extended-statement-name

- LOCAL
- GLOBAL
- 'cursor-name'
- :host-variable
- local-variable
- routine-parameter

Parameters

**statement-name**

Specifies the name of the statement as an identifier.

**'statement-name'**

Specifies the name of the statement as a literal whose value must conform to the rules for an identifier.

**:host-variable**

Specifies the name of the statement as a host-variable whose value must conform to the rules for an identifier.

**local-variable**

Specifies the name of the statement as a local-variable whose value must conform to the rules for an identifier.

**routine-parameter**

Specifies the name of the statement as a routine-parameter whose value must conform to the rules for an identifier.

**LOCAL/GLOBAL**

Specifies the scope of the associated statement name:

- LOCAL indicates that the statement can be referenced only from within the program where it is prepared.
- GLOBAL indicates that the statement can be referenced from any program executing within the same SQL transaction.

The default is LOCAL.
**Usage**

**Static Versus Extended Statement Names**

A static statement name is one coded as a simple identifier. The following PREPARE statement assigns the static name “select1” to the statement being prepared:

```
PREPARE select1 from :select1
```

An extended statement name is one coded either as a literal, a host-variable, a local-variable, or a routine-parameter.

```
MOVE 'SELECT1' to statement-nam
PREPARE :statement-nam FROM :select1-text
```

If a statement is prepared using a static name, it must be referenced using a static name; similarly, if it is prepared using an extended name, it must be referenced using an extended name that has the same scope option.

**Uniqueness of Statement Names**

Static and extended names do not have to be unique with respect to each other. If two statements are assigned the same value for a name, they are considered two separate statements provided that either:

- One of the names is static while the other is extended.
- Both of the names are extended, but they have different scopes, as indicated by their LOCAL/GLOBAL parameter.
Example

Static-statement-name

The following PREPARE statement creates a statement using a static statement name of S1. It is referenced within the subsequent DESCRIBE statement:

```
EXEC SQL
  PREPARE S1 FROM :TEXT
END-EXEC
EXEC SQL
  DESCRIBE S1 USING DESCRIPTOR SQLDA
END-EXEC
```

Extended-statement-name

The following PREPARE statement creates a local statement using an extended statement name of S1. It is then referenced in the subsequent DESCRIBE statement:

```
EXEC SQL
  PREPARE 'S1' FROM :TEXT
END-EXEC
EXEC SQL
  DESCRIBE 'S1' USING DESCRIPTOR SQLDA
END-EXEC
```

Note: Even though S1 is used as the statement name in both of the above examples, two separate statements are created: one with a static name of S1 and one with an extended name of S1.

Global Extended-statement-name

The following PREPARE statement creates a global statement using an extended statement name whose value is not known until runtime. In this case, the value 'S2' is moved to the host variable before the statement is executed and will be the name of the statement created:

```
MOVE 'S2' TO :SNAME
EXEC SQL
  PREPARE GLOBAL :SNAME FROM :TEXT
END-EXEC
```
Since this is a global statement, it can be referenced in a different program than the one where the PREPARE statement appears. For example, the following DESCRIBE statement might be contained in a different program:

```sql
EXEC SQL
   DESCRIBE GLOBAL 'S2'
END-EXEC
```

**Note:** It does not matter that in one case the name of the statement is supplied through a host-variable and in the other it is specified as a literal. They both refer to the global statement S2.
Chapter 3: Data Types and Null Values

This section contains the following topics:

- Data Types (see page 53)
- Expansion of Data-type (see page 55)
- Representation of Date/Time Values (see page 63)
- Comparison, Assignment, Arithmetic, and Concatenation Operations (see page 66)
- Null Values (see page 73)

Data Types

A data type is a set of values that share processing characteristics. For example, the set of all integers is a data type. Every column, local variable, parameter, and host variable has an associated data type.

Data Types and Value Sets

The data type limits the set of values that can occur in the column, local variable, parameter, or host variable. The data type also determines the operations that can be performed on values in the column, local variable, parameter, or host variable.

You associate a data type with the following:

- A column when you define the column
- An SQL routine local variable when you define the local variable
- An SQL-invoked routine parameter when you define the parameter
- A host variable when you declare the host variable

More Information

- For more information about defining columns, see CREATE TABLE and ALTER TABLE.
- For more information about declaring host variables, see Host Variables.
- For more information about declaring local variables, see Local Variables.
- For more information about declaring routine-parameters, see Routine Parameters.
Categories of Data Types

CA IDMS supports the following data types:

<table>
<thead>
<tr>
<th>Category</th>
<th>Data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approximate numeric</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>REAL</td>
</tr>
<tr>
<td>Binary</td>
<td>BINARY</td>
</tr>
<tr>
<td>Character string</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>VARCHAR (or CHAR VARYING)</td>
</tr>
<tr>
<td>Date/time</td>
<td>DATE</td>
</tr>
<tr>
<td></td>
<td>TIME</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>Exact numeric</td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LONGINT (or BIGINT)</td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
</tr>
<tr>
<td></td>
<td>UNSIGNED DECIMAL</td>
</tr>
<tr>
<td></td>
<td>UNSIGNED NUMERIC</td>
</tr>
<tr>
<td>Graphics character string</td>
<td>GRAPHIC</td>
</tr>
<tr>
<td></td>
<td>VARGRAPHIC</td>
</tr>
<tr>
<td>XML data</td>
<td>XML</td>
</tr>
</tbody>
</table>

More Information

- For more information about a description of the XML data type, see XML Data Type and XML Values.
- For more information about a description of all the other data types, see Expansion of Data-type.
Determining the Data Type of a Value

The data type of a value in a column, host variable, local variable, or parameter is the data type of the column, host variable, local variable or routine parameter. For example, every value in a column with a data type of INTEGER has a data type INTEGER.

The literal used to represent a value must be appropriate for the data type of the value. For example, 983 represents a numeric value; '983' represents a character value.

Note: For more information about literals, see Literals (see page 75).

Data Types Effect on Processing

The data type of a value determines:

- Columns, local variables, parameters, and host variables to which it can be assigned
- Values with which it can be compared
- Operations where it can be used
- Results of operations where it is combined with values of other data types
- Its internal representation and storage requirements

More information:

ALTER TABLE (see page 290)
CREATE TABLE (see page 378)
Local Variables (see page 81)
Expansion of Data-type (see page 55)
Host Variables (see page 77)
Routine Parameters (see page 84)

Expansion of Data-type

The expanded parameters of data-type specify data types in an SQL data description statement.
Expansion of Data-type

Syntax

![Expansion of data-type diagram]

Parameters

**BINARY**

Identifies a set of values that are fixed-length bit strings. BINARY values are represented by hexadecimal literals (for example, X'12A5E978').

**length**

Specifies the number of eight-bit bytes in a BINARY value. Length must be an integer in the range 1 through 32,760. The default is 1.

The maximum length of a column with a data type of BINARY is limited by page size and the total length of other columns in the table.

**Note:** For more information about the length of a BINARY value, see CREATE TABLE. (see page 378)
CHARacter

Identifies a set of values that are fixed-length single-byte character strings. CHARACTER values are represented by character string literals (for example, 'Past due').

length

Specifies the number of bytes in a CHARACTER value. Length must be an integer in the range 1 through 32,760. The default is 1.

The maximum length of a column with a data type of CHARACTER is limited by page size and the total length of other columns in the table.

Note: For more information about the length of a CHARACTER value, see CREATE TABLE. (see page 378)

DATE

Identifies the set of values that represent valid dates from January 1, 0001, through December 31, 9999. DATE values are represented by character string literals (for example, '1999-07-22').

The maximum length of a DATE value is 10 bytes. Internally, the length of a DATE value is always eight bytes.

DECimal

Identifies a set of fixed-point, signed packed decimal values. DECIMAL values are represented by exact numeric literals (for example, 17.23).

The range of values included in the set is determined by the precision and scale specified for the data type. The largest possible DECIMAL value is 1031-1. The smallest possible DECIMAL value is -(1031-1).

precision

Specifies the number of digits in a DECIMAL value. Precision must be an integer in the range 1 through 56. The default is 56.

The length of a DECIMAL value is equal to the precision plus 1, divided by 2.

scale

Specifies the number of digits to the right of the decimal point in a DECIMAL value. Scale must be an integer in the range 0 through the precision of the DECIMAL value. The default is 0.
DOUBLE PRECISION

Identifies the set of 64-bit (long) floating-point values with a seven-bit exponent and a binary precision of 56. DOUBLE PRECISION values are represented by approximate numeric literals (for example, 0.1E-16).

The magnitude that can be represented by a positive DOUBLE PRECISION value ranges from approximately 5.4E-79 to approximately 7.2E+75. The magnitude that can be represented by a negative DOUBLE PRECISION value ranges from approximately -5.4E-79 to approximately -7.2E+75.

The length of a DOUBLE PRECISION value is eight bytes.

FLOAT

Identifies a set of floating-point values with a seven-bit exponent and a user-specified precision. FLOAT values are represented by approximate numeric literals (for example, -1.4E9).

The magnitude that can be represented by a positive FLOAT value ranges from approximately 5.4E-79 to approximately 7.2E+75. The magnitude that can be represented by a negative FLOAT value ranges from approximately -5.4E-79 to approximately -7.2E+75.

\textit{precision}

Specifies the binary precision of a FLOAT value. \textit{Precision} must be an integer in the range 1 through 56. The default is 24.

If \textit{precision} is less than or equal to 24, the length of a FLOAT value is four bytes. If \textit{precision} is greater than 24, the length of a FLOAT value is eight bytes.

GRAPHIC

Identifies a set of values that are fixed-length double-byte character strings. GRAPHIC values are represented by double-byte character string literals (for example, G'<####>', where < and > represent the shift-out and shift-in characters and # represents a double-byte character).

The GRAPHIC data type is a CA IDMS extension of the SQL standard.

\textit{length}

Specifies the number of characters in a GRAPHIC value. The length in bytes of a GRAPHIC value is equal to the number of bytes in one character times the number of characters.

\textit{Length} must be an integer in the range 1 through 16,380. The default is 1.

The maximum length of a column with a data type of GRAPHIC is limited by page size and the total length of other columns in the table.

\textbf{Note:} For more information about the length of a GRAPHIC value, see \texttt{CREATE TABLE} (see page 378).
**INTeger**

Identifies the set of values that are 31-bit signed integers in the range -2,147,483,648 through 2,147,483,647. INTEGER values are represented by exact numeric literals (for example, -2874).

The length of an INTEGER value is four bytes.

**BIGINT (or LONGINT)**

Identifies the set of values that are 63-bit signed integers in the range -9,223,372,036,854,775,808 through 9,223,372,036,854,775,807. BIGINT values are represented by exact numeric literals (for example, 2187168).

The length of a BIGINT value is eight bytes. The keyword LONGINT can be used as a synonym for BIGINT but this is a CA IDMS extension of the SQL standard.

**NUMeric**

Identifies a set of fixed-point, signed zoned decimal values. NUMERIC values are represented by exact numeric literals (for example, -4.7). The use of NUM as a synonym for NUMERIC is a CA IDMS extension of the SQL standard.

The range of values included in the set is determined by the precision and scale specified for the data type. The largest possible NUMERIC value is 10^{31} - 1. The smallest possible NUMERIC value is -(10^{31} - 1).

**precision**

Specifies the number of digits in a NUMERIC value. Precision must be an integer in the range 1 through 31. The default is 1.

The length in bytes of a NUMERIC value is equal to the precision.

**scale**

Specifies the number of digits to the right of the decimal point in a NUMERIC value. Scale must be an integer in the range 0 through the precision of the NUMERIC value. The default is 0.
REAL

Identifies the set of 32-bit (short) floating-point values with a seven-bit exponent and a binary precision of 24. REAL values are represented by approximate numeric literals (for example, 0.4E52).

The magnitude that can be represented by a positive REAL value ranges from approximately 5.4E-79 to approximately 7.2E+75. The magnitude that can be represented by a negative REAL value ranges from approximately -5.4E-79 to approximately -7.2E+75.

The length of a REAL value is four bytes.

SMALLINT

Identifies the set of values that are 15-bit signed integers in the range -32,768 through 32,767. SMALLINT values are represented by exact numeric literals (for example, 16433).

The length of a SMALLINT value is two bytes.

TIME

Identifies the set of values that represent valid times from 00.00.00 through 23.59.59.

TIME values are represented by character string literals (for example, '13.42.59').

The maximum length of a TIME value is eight bytes. Internally, the length of a TIME value is always eight bytes.

Note: A TIME value of 24.00.00 is accepted and treated as 00.00.00.

TIMESTAMP

Identifies the set of values that represent valid date/time combinations with a precision of millionths of a second. Valid dates range from January 1, 0001, through December 31, 9999. Valid times range from 00.00.00.000000 through 23.59.59.999999.

TIMESTAMP values are represented by character string literals (for example, '1999-05-02-09.46.39.738294').

The maximum length of a TIMESTAMP value is 26 bytes. Internally, the length of a TIMESTAMP value is always eight bytes.
UNSIGNED DECIMAL

Identifies a set of fixed-point, unsigned packed decimal values. UNSIGNED DECIMAL values are represented by exact numeric literals (for example, 17.23).

The range of values included in the set is determined by the precision and scale specified for the data type. The largest possible UNSIGNED DECIMAL value is \(10^{31}-1\). The smallest possible DECIMAL value is 0.

The UNSIGNED DECIMAL data type is a CA IDMS extension of the SQL standard.

**precision**

Specifies the number of digits in an UNSIGNED DECIMAL value. *Precision* must be an integer in the range 1 through 31. The default is 1.

The length of an UNSIGNED DECIMAL value is equal to the precision plus 1, divided by 2.

**scale**

Specifies the number of digits to the right of the decimal point in an UNSIGNED DECIMAL value. *Scale* must be an integer in the range 0 through the precision of the UNSIGNED DECIMAL value. The default is 0.

UNSIGNED NUMERIC

Identifies a set of fixed-point, unsigned zoned decimal values. UNSIGNED NUMERIC values are represented by exact numeric literals (for example, 4.7).

The range of values included in the set is determined by the precision and scale specified for the data type. The largest possible UNSIGNED NUMERIC value is \(10^{31}-1\). The smallest possible NUMERIC value is 0.

The UNSIGNED NUMERIC data type is a CA IDMS extension of the SQL standard.

**precision**

Specifies the number of digits in an UNSIGNED NUMERIC value. *Precision* must be an integer in the range 1 through 31. The default is 1.

The length in bytes of an UNSIGNED NUMERIC value is equal to the precision.

**scale**

Specifies the number of digits to the right of the decimal point in an UNSIGNED NUMERIC value. *Scale* must be an integer in the range 0 through the precision of the NUMERIC value. The default is 0.
**VARCHAR (or CHAR VARYING)**

Identifies a set of values that are variable-length single-byte character strings. VARCHAR values are represented by character string literals (for example, 'Customer address needs to be verified').

**length**

Specifies the maximum number of characters in a VARCHAR value. Length must be an integer in the range 1 through 32,758. The default is 1.

The length of a VARCHAR value is the number of characters in the value. The number of bytes reserved for a VARCHAR value is always the same; the maximum length, plus 2 regardless of the length of the VARCHAR value. A VARCHAR value is preceded by a 2-byte binary length of the value.

The maximum length of a column with a data type of VARCHAR is limited by page size, the total length of other columns in the table, and other factors.

**Note:** For more information about the length of a VARCHAR value, see CREATE TABLE. (see page 378)

**VARGRAPHIC**

Identifies a set of values that are variable-length double-byte character strings. VARGRAPHIC values are represented by double-byte character string literals (for example, G'<####>', where < and > represent the shift-out and shift-in characters and # represents a double-byte character).

The VARGRAPHIC data type is a CA IDMS extension of the SQL standard.

**length**

Specifies the maximum number of characters in a VARGRAPHIC value. Length must be an integer in the range 1 through 16,379. The default is 1.

The length of a VARGRAPHIC value is the number of characters in the value. The number of bytes reserved for a VARGRAPHIC value is the maximum length times the number of bytes for one character, plus 2. A VARGRAPHIC value is preceded by a 2-byte binary length of the value.

The maximum length of a column with a data type of VARGRAPHIC is limited by page size and the total length of other columns in the table.

**Note:** For more information about the length of a VARGRAPHIC value, see CREATE TABLE. (see page 378)

**Usage**

**Graphics Data**

The use of graphics data requires the installation of CA IDMS DBCS.
Example

Defining Table Columns

The following CREATE TABLE statement creates a table with ten columns. Each column is associated with a data type. The data type specification determines the set of values that can occur in the column.

create table job
  (job_id           integer        not null,
   job_title         character(20)  not null,
   job_desc_line_1   varchar(60),
   job_desc_line_2   varchar(60),
   min_rate          decimal(8,2),
   max_rate          decimal(8,2),
   salary_ind        character(1),
   num_of_positions  smallint,
   num_open          smallint,
   eff_date          date);

Representation of Date/Time Values

Values whose data types are DATE, TIME, or TIMESTAMP are represented internally as binary numbers and externally as character strings.

Internal Representation

Date/time values are represented internally by 64-bit binary numbers. The bits, numbered from the left starting with 0, have the following meaning:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 0 through 26</td>
<td>Number of days since January 1, 0001</td>
</tr>
<tr>
<td>Bits 27 through 43</td>
<td>Number of seconds in the day since midnight</td>
</tr>
<tr>
<td>Bits 44 through 63</td>
<td>Number of microseconds since the last second</td>
</tr>
</tbody>
</table>
External Representations

Date/time Representations

CA IDMS provides four standard external representations for a date/time value:

- International Standards Organization (ISO)
- IBM USA standard (USA)
- IBM European standard (EUR)
- Japanese Industrial Standard Christian Era (JIS)

You can reference the abbreviation associated with the external representation (for example, ISO) when you use the CHAR function.

**Note:** For more information about the CHAR function, see CA IDMS Scalar Functions, (see page 124)

Date Values

The external representation of a date value is a character string which begins with a digit and must be at least five characters. The representation of a date value can:

- Omit leading zeros from the month, day, and year portions
- Include trailing blanks

The following table describes the external representations of a date value:

<table>
<thead>
<tr>
<th>Standard</th>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO</td>
<td>yyyy-mm-dd</td>
<td>1990-12-15</td>
</tr>
<tr>
<td>USA</td>
<td>mm/dd/yyyy</td>
<td>12/15/1990</td>
</tr>
<tr>
<td>EUR</td>
<td>dd.mm.yyyy</td>
<td>15.12.1990</td>
</tr>
<tr>
<td>JIS</td>
<td>yyyy-mm-dd</td>
<td>1990-12-15</td>
</tr>
</tbody>
</table>
External Representations of Time Values

The external representation of a time value is a character string which begins with a digit and must be at least five characters. The representation of a time value may omit a leading zero from the hours, minutes, and seconds portions and may include trailing blanks.

The following table describes the external representations of a time value:

<table>
<thead>
<tr>
<th>Standard</th>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO</td>
<td>hh.mm.ss</td>
<td>16.43.17</td>
</tr>
<tr>
<td>USA</td>
<td>hh:mm AM</td>
<td>4:43 PM</td>
</tr>
<tr>
<td></td>
<td>hh:mm PM</td>
<td></td>
</tr>
<tr>
<td>EUR</td>
<td>hh.mm.ss</td>
<td>16.43.17</td>
</tr>
<tr>
<td>JIS</td>
<td>hh:mm:ss</td>
<td>16:43:17</td>
</tr>
</tbody>
</table>

External Representations of Timestamps

The external representation of a timestamp is a character string which begins with a digit and must be at least 11 characters. The timestamp format is:

```
yyyy-mm-dd-hh.mm.ss.nnnnnn
```

With the representation of a timestamp value:

- Omit leading zeros from the year, month, day, hours, minutes, and second portions
- Truncate or omit microseconds entirely (any digit of microseconds that is omitted is assumed to be zero)
- Include trailing blanks
Entering and Retrieving Date/Time Values

When you enter date/time values into the database or use them in predicates, you can use any of the formats documented above. CA IDMS determines the format by examining the value.

When you retrieve date/time values from the database, the format in which the value is retrieved depends on how the SQL statement is issued:

- If the statement is embedded in a program, the format is determined by the DATE and TIME precompiler options, if specified, or by the installation default.
- If the statement is submitted through the Command Facility, all date/time values are returned in ISO format.

Note: For more information about precompiler options, see the CA IDMS SQL Programming Guide.

Comparison, Assignment, Arithmetic, and Concatenation Operations

The data type, length, and magnitude of a value determine how CA IDMS handles the value during comparison, assignment, arithmetic, and concatenation operations. The same factors also affect the outcome of these operations.

For example:

- Padding can occur during comparisons of values of unequal length.
- Overflow occurs when the magnitude of a numeric value is greater than the largest magnitude represented by the data type of the construct to which the value is assigned.
- Underflow occurs when the magnitude of an approximate numeric value is smaller than the smallest magnitude represented by the data type of the construct to which the value is assigned.
- Truncation can occur when a binary, graphic, or character value is assigned to a construct that is not big enough to hold the value.
- Rounding can occur as a result of truncation during assignment and arithmetic operations.
Binary Values

Comparison

Binary values can be compared to the following:
- Other binary values. When comparing binary values of different lengths, CA IDMS pads the shorter value with binary zeros on the right to make the lengths equal.
- Date/time values. Provided the binary value has a length of 8.
- Character values. If the binary and character values are of different lengths, CA IDMS pads the shorter value with spaces on the right to make the lengths equal.

Assignment

Binary values can be assigned to the following:
- Binary constructs (for example, a column with a data type of BINARY):
  - If the binary value is shorter than the construct, CA IDMS pads the value with binary zeros on the right to make the value as long as the construct.
  - If the binary value is longer than the construct and the construct is a column (for example, in an INSERT or UPDATE operation), CA IDMS truncates the value on the right if the portion to be truncated contains only binary zeros. If nonzero bits would be truncated, CA IDMS generates an exception and no assignment occurs.
  - If the binary value is longer than the construct and the construct is a host variable (for example, in a FETCH or SELECT operation), CA IDMS truncates the value on the right.
- Date/time constructs (for example, a column with a data type of TIMESTAMP), provided the binary value has a length of 8 and conforms to the internal representation of a valid date/time value.
- Character constructs (for example, a column with a data type of CHARACTER). The rules for assigning a binary value to a character construct are the same as those for assigning a character value to a character construct. If padding is necessary, it is done with blanks instead of binary zeros.

Arithmetic

You cannot use binary values in arithmetic operations.

Concatenation

Binary values can be concatenated with other binary values and with character values. For the purposes of concatenation, binary values are treated as character values, and the result of a concatenation operation is a character value.
Character Values

Comparison

Character values can be compared to the following:

- Other character values. When comparing character values of different lengths, CA IDMS pads the shorter value with blanks on the right to make the lengths equal.
- Date/time values, provided the character value has the format of a date/time value of the same data type as the date/time value used in the comparison.
- Binary values. The binary value is treated as a character value and padded with blanks, if necessary, before the comparison is made.

Assignment

Character values can be assigned to the following:

- Character constructs (for example, a column with a data type of CHARACTER or VARCHAR). If the character value is:
  - Shorter than the construct, and the construct has a data type of CHARACTER, CA IDMS pads the value with blanks on the right to make the value as long as the construct. If the construct has a data type of VARCHAR, no padding takes place.
  - Longer than the construct and the construct is a column (for example, in an INSERT or UPDATE operation), CA IDMS truncates the value on the right if all the characters to be truncated are blanks. If nonblank characters would be truncated, CA IDMS generates an exception and no assignment occurs.
  - Longer than the construct and the construct is a host variable, a local variable, or a parameter (for example, in a FETCH, SELECT, or a SET operation), CA IDMS truncates the value on the right and an SQL warning condition is issued.
- Date/time constructs (for example, a column with a data type of TIMESTAMP), provided the character value has the format of a date/time value of the same data type as the construct to which the value is being assigned.
- Binary constructs (for example, a column with a data type of BINARY). The rules for assigning a character value to a binary construct are identical to those for assigning a binary value to a binary construct. If padding is necessary, it is done with binary zeros instead of blanks.
Arithmetic

You cannot use character values in arithmetic operations.

Concatenation

Character values can be concatenated with other character values.

If one value in the concatenation:

- Has a data type of VARCHAR, the result has a data type of VARCHAR. Otherwise, the result has a data type of CHARACTER.
- Is null, the result is null. Otherwise, the length of the result is the sum of the lengths of the two values that are concatenated. The length of the result cannot exceed 32,760.

A character value and a binary value can be concatenated. The binary value is treated as a character value for the purpose of concatenation.

Date/time Values

Comparison

Date/time values can be compared to:

- Other date/time values of the same data type
- Binary values, provided the binary value has a length of 8
- Character values, provided the character value has the format of a date/time value of the same data type as the date/time value used in the comparison

Assignment

Date/time values can be assigned to the following:

- Date/time constructs of the same data type. For example, a value with a data type of TIMESTAMP can be assigned to a column with a data type of TIMESTAMP.
- Binary constructs with a length of 8.
- Character constructs long enough to contain the date/time value

If the date/time value is:

- Shorter than the character construct, CA IDMS pads the value with blanks on the right to make the value as long as the construct
- Longer than the character construct, CA IDMS generates an exception and does not perform the assignment
Arithmetic

You can use date/time values only in addition and subtraction. Special rules govern date/time arithmetic.

Note: For information about date/time arithmetic, see Date/time Arithmetic. (see page 110)

Concatenation

You can concatenate a date value and a time value using the TIMESTAMP function.

Note: For more information about the TIMESTAMP function, see CA IDMS Scalar Functions (see page 124).
Graphics Character Values

Comparison

Graphics character values can be compared to other graphics character values. When comparing graphics character values of different lengths, CA IDMS pads the shorter value with double-byte blank characters on the right to make the lengths equal.

Assignment

Graphics character values can be assigned to graphics character constructs (for example, to a column with a data type of GRAPHIC or VARGRAPHIC): If the graphics character value is:

- Shorter than the construct, and the construct has a data type of GRAPHIC, CA IDMS pads the value with double-byte blank characters on the right to make the value as long as the construct. If the construct has a data type of VARGRAPHIC, no padding takes place.
- Longer than the construct and the construct is a column (for example, in an INSERT or UPDATE operation), CA IDMS truncates the value on the right if the portion to be truncated contains only double-byte blank characters. If nonblank characters would be truncated, CA IDMS generates an exception and no truncation occurs.
- Longer than the construct and the construct is a host variable (for example, in a FETCH or SELECT operation), CA IDMS truncates the value on the right.

Arithmetic

You cannot use graphics character values in arithmetic operations.

Concatenation

Graphics character values can be concatenated with other graphics character values.

If one of the values has a data type of VARGRAPHIC, the result has a data type of VARGRAPHIC. Otherwise, the result has a data type of GRAPHIC.

If one value in the concatenation is null, the result is null. Otherwise, the length of the result is the sum of the lengths of the two values that are concatenated. The length of the result cannot exceed 32,760.

Usage

The use of graphics data requires the installation of CA IDMS DBCS.
Numeric Values

Comparison

Numeric values can be compared only to other numeric values.

Assignment

Numeric values can be assigned only to numeric constructs (for example, to a column with a data type of DECIMAL).

When assigning a value of one numeric data type to a construct of a different numeric data type, CA IDMS converts the value to the data type of the construct:

- If the conversion results in overflow or underflow, CA IDMS generates an exception and does not perform the assignment.
- If the conversion results in the loss of digits to the right of the decimal point in an exact numeric value or least significant digits in the mantissa of an approximate numeric value, CA IDMS rounds the value and does not generate an exception.

Arithmetic

You can use values of all numeric data types (both approximate and exact) in arithmetic operations. Additionally, a single arithmetic expression can include values of more than one numeric data type.

Data Type Conversion for Comparison and Arithmetic

When a comparison or arithmetic operation involves two values of different numeric data types, CA IDMS determines which data type has higher precedence. CA IDMS then converts both values to a common data type based on the data type of higher precedence:

<table>
<thead>
<tr>
<th>Data type of highest precedence</th>
<th>Common data type for conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td></td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>NUMERIC</td>
<td></td>
</tr>
<tr>
<td>LONGINT</td>
<td></td>
</tr>
</tbody>
</table>
Null Values

A null value is a placeholder that indicates the absence of a value. Null values exist for all data types. The null value of a given data type is different from all non-null values of the same data type.

By default, any column can contain null values. You can use either the NOT NULL or the CHECK parameter in a column definition to disallow null values in the column.

Parameters and local variable of SQL-invoked routines can always contain null values. However, it is possible to define initial values.

Host variables can also represent null values. You use an indicator variable with a host variable to indicate whether the host variable represents a null value.

---

### Data Type of Highest Precedence

<table>
<thead>
<tr>
<th>Lowest</th>
<th>INTEGER</th>
<th>SMALLINT</th>
<th>INTEGER</th>
</tr>
</thead>
</table>

### Precision of the Result

The precision of the result of comparison or arithmetic operation when a conversion involves a decimal data type is:

- The maximum number of digits to the left of the decimal in the source operands plus the result scale plus 1, if the operation is addition or subtraction (The result scale is the maximum of the source scales)
  
  For example, the precision of the result of 45673 + 5.398 is 9.

- The sum of the precision of the first value and the second value, if the operation is multiplication or division
  
  For example, the precision of the result of 45 x 367 is 5.

### Examples of Data Type Conversion

For example, to add a NUMERIC value to an INTEGER value, CA IDMS converts both values to DECIMAL. To compare a FLOAT value to a REAL value, CA IDMS converts both values to DOUBLE PRECISION.
How You Specify a Null Value

You use the keyword NULL to indicate a null value. For example, the following INSERT statement inserts a new row into the DEPARTMENT table. The department number and name and the division code are known, but the department head has not been appointed yet. A null value is used as a placeholder in the DEPT_HEAD_ID column.

```sql
insert into department
    (dept_id, dept_name, div_code, dept_head_id)
values (4040, 'Audit', 'D09', null);
```

Null Values in Comparison and Arithmetic Operations

Null values have the following effect in comparison and arithmetic operations:

- The result of a **comparison operation** involving one or more null values is always unknown
- The result of an **arithmetic operation** involving one or more null values is always a null value

Null Values in Sort Operations

In a sort operation, a null value is a high value. Thus, a null is placed at the end of an ascending sort sequence.

More Information

- For more information about defining columns, see CREATE TABLE or CREATE TEMPORARY TABLE.
- For more information about CREATE FUNCTION, CREATE PROCEDURE, and compound statements, see Statements.
- For more information about indicator variables, see Indicator Variables.

More information:

- [CREATE FUNCTION](see page 341)
- [CREATE PROCEDURE](see page 361)
- [CREATE TABLE](see page 378)
- [CREATE TEMPORARY TABLE](see page 392)
- [Statements](see page 249)
Chapter 4: Values and Value Expressions

This section contains the following topics:

- **Literals** (see page 75)
- **Expansion of Literal** (see page 75)
- **Host Variables** (see page 77)
- **Expansion of Host-variable** (see page 79)
- **Local Variables** (see page 81)
- **Expansion of Local-variable** (see page 81)
- **Routine Parameters** (see page 84)
- **Expansion of Routine-parameter** (see page 85)
- **Dynamic Parameters** (see page 87)
- **Expansion of Dynamic-parameter-marker** (see page 92)
- **Special Registers** (see page 93)
- **Expansion of Special-register** (see page 93)
- **ROWID Pseudo-column** (see page 95)
- **Expansion of rowid-pseudo-column** (see page 96)
- **Expansion of Value-expression** (see page 101)
- **Durations** (see page 107)
- **Expansion of Labeled-duration** (see page 107)
- **Date/time Arithmetic** (see page 110)
- **Expansion of XML-value-expression** (see page 116)

**Literals**

A literal directly represents a specific value. CA IDMS uses several types of literals. Each type of literal represents values of one or more data types. For example, a character string literal represents either a CHARACTER value or a VARCHAR value.

**Note:** For more information about data types, see Values and Value Expressions.

**More information:**

Values and Value Expressions (see page 75)

**Expansion of Literal**

The expanded parameters of literal represent specific data values in an SQL statement.
Expansion of Literal

Syntax

Expansion of literal

`'character-string-literal'`  
G'double-byte-character-string-literal'  
X'hexadecimal-literal'  
exact-numeric-literal  
approximate-numeric-literal

Parameters

`'character-string-literal'`

Represents a character value as a string of single-byte characters (for example, '79 High Street').

A character string literal can consist of any combination of letters, digits, and special characters (including blanks). Lowercase letters are not equal to uppercase letters in a character string literal (for example, 'Boston' is not equal to 'BOSTON').

Character string literals must be enclosed in single quotation marks. To include a single quotation mark as part of the character string itself, code two consecutive single quotation marks (for example, 'Carol''s job').

G'double-byte-character-string-literal'

Represents a character value as a string of double-byte characters. Within a double-byte character string literal, a sequence of one or more double-byte characters must be preceded by the shift-out character and followed by the shift-in character (for example, G'<#####>', where < and > represent the shift-out and shift-in characters and # represents a double-byte character). The shift characters are not part of the data value.

A double-byte character string literal can consist of any combination of characters in the double-byte character set (including the double-byte blank character). The entire sequence of characters, including the shift-out and shift-in characters, must be enclosed in single quotation marks.

Note: Certain hardware configurations do not require the use of shift characters in double-byte character string literals.

Note: 'SQL Standard Compatibility'. Double-byte character string literals are a CA IDMS extension of the SQL standard.
**X'hexadecimal-literal'**

Represents a binary value as a sequence of an even number of hexadecimal digits (for example, X'01F27A'). Hexadecimal literals must be enclosed in single quotation marks.

Case is not significant in hexadecimal literals. You can use either uppercase or lowercase for the digits A through F.

**exact-numeric-literal**

Represents a signed or unsigned fixed-point decimal number. The decimal point in an exact numeric literal can be either explicit (for example, 39523.142) or implicit after the rightmost digit (for example, -2834).

**approximate-numeric-literal**

Represents a floating point number. Approximate numeric literals have the form `mantissaEexponent` (for example, 3.45E-2), where:

- **Mantissa** is an exact numeric literal
- **Exponent** is a signed or unsigned integer

**Example**

**Specifying Values for a New Row**

The INSERT statement below inserts a new row into the COVERAGE table. The values in the row are represented by the following types of literals: character string (first column), exact numeric (second and fourth columns), and date (third column).

```
insert into coverage (plan_code, emp_id, selection_date, num_dependents)
values ('002', 2538, '1989-05-11', 3);
```

**Host Variables**

A host variable is a variable that is referenced in an SQL statement embedded in an application program. You use host variables to:

- Make data in the CA IDMS database available for processing by an application program
- Make data from sources other than the database (for example, from a sequential file) available for processing by SQL statements

Host variables are not used in SQL statements submitted through the command facility.

**Note:** An INTO clause is required for SQL SELECT statements embedded in host programs.
Indicator Variables

An indicator variable is a host variable used to indicate whether the value in another host variable represents a null or truncated value. The use of indicator variables is optional. You can reference a host variable in an SQL statement with or without naming an associated indicator variable. However, if a SELECT or FETCH statement causes a null value to be assigned to a host variable without an associated indicator variable, CA IDMS returns an error.

Indicator Variable Values

When assigning a value to a host variable, CA IDMS sets the associated indicator variable as follows:

<table>
<thead>
<tr>
<th>Indicator variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>The value assigned to the host variable was null. The contents of the host variable are unchanged.</td>
</tr>
<tr>
<td>0</td>
<td>The host variable contains a non-null value that has not been truncated.</td>
</tr>
<tr>
<td>1 or greater</td>
<td>The host variable contains a truncated value. The value in the indicator variable is the length in bytes of the original untruncated value.</td>
</tr>
</tbody>
</table>

Declaring Host Variables

Before you execute an SQL statement that references a host variable, you must declare the variable to CA IDMS:

- Explicitly in an SQL declaration section
- Implicitly with an INCLUDE statement

A single application program can include both explicit and implicit host variable declarations.

Note: For more information about using the INCLUDE statement to declare host variables, see INCLUDE. (see page 479)
SQL Declaration Section

In an explicit host variable declaration, you specify the name and data type of the variable. Depending on the program language, additional information about the variable may also be required (for example, the COBOL level number).

Host variable names must conform to language-specific rules for forming variable names.

**Note:** For more information about language-specific instructions for declaring host variables in an SQL declaration section, see the *CA IDMS SQL Programming Guide*.

**More information:**

[INCLUDE](see page 479)

Expansion of Host-variable

The expanded parameters of host-variable identify declared program variables.

**Syntax**

```
Expansion of host-variable

:host-variable-name
  indicator :indicator-variable-name
```

**Parameters**

`:host-variable-name`

Identifies the name of a host variable previously declared to the program.

`indicator :indicator-variable`

Identifies the name of a host variable previously declared to the program to serve as an indicator variable.

*Indicator-variable* must reference a variable that was declared with a numeric data type.
Expansion of Host-variable

Usage

A Colon Must Precede the Variable Name

When you reference a host variable in an SQL statement, you must precede the variable name with a colon (:).

No Comma Between Host and Indicator Variables

There is no required separator between a host variable and its associated indicator variable when they are referenced in an SQL statement. The only valid separator is the optional keyword indicator.

Non-bulk Structure in COBOL

In certain SQL statements embedded in a COBOL application program, host-variable may refer to a non-bulk structure defined in the host program.

Note: For more information about commas as separators, see the CA IDMS SQL Programming Guide.

Example

The following SELECT statement retrieves data into host variables. BIRTH-DATE-I is an indicator variable that is associated with BIRTH-DATE-HOST because the BIRTH_DATE column may contain null values.

EXEC SQL
  SELECT DEPT_ID, DEPT_NAME, EMP_ID
  INTO :DEPT-ID-HOST,
       :DEPT-NAME-HOST,
       :EMP-ID-HOST,
       :BIRTH-DATE-HOST :BIRTH-DATE-I
  FROM DEPARTMENT D, EMPLOYEE E
  WHERE D.DEPT_ID = E.DEPT_ID
END-EXEC
Local Variables

A local variable is a variable that is defined in an SQL routine. You use local variables to temporarily store and retrieve values as needed in the logic of the routine. Local variables are used for such things as:

- Retrieving data from a CA IDMS database by specifying them on the INTO clause of a SELECT statement
- Passing data to and from other SQL-invoked routines by specifying them as arguments on the routine invocation
- Holding computational values by specifying them as a target of a SET statement or as values within expressions.

Local variables can only be referenced within the body of the SQL routine in which they are defined.

Declaring Local Variables

A local variable is defined by a variable-declaration statement that is included in a compound statement within an SQL routine body. The declaration of a local variable consists of the specification of its name, data type, and optionally its initial value.

**Note:** For more information about declaring local variables, see [Compound Statement](#) (see page 566)

**More information:**

[Compound Statement](#) (see page 566)

Expansion of Local-variable

The expanded parameters of local-variable identify program variables declared in a compound statement.

**Syntax**

```
Expansion of local-variable
```

```
local-variable-name
```

```
cmp-stmt-label.
```

```
Parameters

cmp-stmnt-label

Specifies the label of the compound statement that contains the definition of local-variable.

local-variable-name

Identifies the local variable of an SQL routine.

Usage

Referencing Local Variables

A local variable can only be referenced from within the compound statement that contains its declaration or from within a compound statement contained in the compound statement that contains its declaration.

Avoiding Ambiguous References

The name of a local variable of an SQL routine can be the same as the name of another local variable, a routine parameter, a column, or another schema-defined entity such as a table. To avoid ambiguity when referencing these objects, qualification can be used as follows:

- A local variable can be qualified with the label of the compound statement in which it is declared.
- A routine parameter can be qualified with its associated schema and routine name.
- A column can be qualified with its schema and table name.
- Other schema-defined objects can be qualified with the name of the schema in which they are defined.
Resolving Ambiguous References

If a name is not qualified and more than one object has the specified name, CA IDMS uses the following precedence rules to resolve the ambiguous reference:

- If a local variable with a matching name has been declared within the compound statement in which the reference occurs, the reference is to the local variable. If more than one such variable is declared, the reference is to the variable declared in the innermost compound statement containing the reference.

- If a parameter of the routine in which the reference occurs has a matching name, the reference is to the routine parameter.

- Otherwise, the reference is treated as a reference to a schema-defined object.

  **Note:** For more information about how such a reference is resolved, see Resolving References to Entities in Schemas.

In the SQL standard, an unqualified reference would be to the object with innermost scope.
Example

In the following SQL procedure, two local variables, FNAME and LNAME are defined. The references are qualified in the SELECT statement with the label of the compound statement that holds the definition of the local variables. The SET statement uses unqualified references.

```sql
set options command delimiter '++';
create procedure SQLROUT.LOCALVAR
  ( TITLE     varchar(10) with default
    , P_EMP_ID  NUMERIC(4)
    , P_NAME    varchar(25)
  )
  external name LOCALVAR language SQL
L_MAIN: begin not atomic
  /*
  ** Count number of employees with equal Firstname using REPEAT
  */
  declare FNAME   char(20);
  declare LNAME   varchar(20);

  select EMP_FNAME, EMP_LNAME
      into L_MAIN.FNAME, L_MAIN.LNAME
    from DEMOEMPL.EMPLOYEE
    where EMP_ID = P_EMP_ID;

  set P_NAME = FNAME || LNAME;
end L_MAIN
++
*+ TITLE       P_EMP_ID  P_NAME
call SQLROUT.LOCALVAR('LOCALVAR',2010)++
++

*+ ......        ..........         ........
*+ LOCALVAR 2010 Cora Parke
```

Routine Parameters

A routine parameter is a parameter of an SQL routine. You use routine parameters to perform the following:

- Pass values to and from the SQL routine
- Store and retrieve values as needed by the routine logic
- Pass values to other SQL-invoked routines

Routine parameters can only be referenced within the body of the SQL routine in which they are defined.
Defining Routine Parameters

A routine parameter is defined through a **parameter-definition** clause of the CREATE PROCEDURE or CREATE FUNCTION statements. The definition includes the specification of the name, the data type and optional WITH DEFAULT attribute.

**Note:** For more information about defining routine parameters, see **CREATE PROCEDURE** (see page 361) and **CREATE FUNCTION**. (see page 341)

More information:

**CREATE FUNCTION** (see page 341)
**CREATE PROCEDURE** (see page 361)

Expansion of Routine-parameter

The expanded parameters of routine-parameter identify routine parameters of an SQL routine.

Syntax

```
Expansion of routine-parameter

─┬──────────────────────────────────────────────┬──
   │ parameter-name                            │
   └─┬───────────────────┬───────
       │ routine-name.                           │
       └─.schema.
```

Parameters

**schema**

Specifies the schema with which the SQL routine identified by **routine-name** is associated.

**routine-name**

Specifies the name of the SQL routine in which the routine parameter identified by **routine-parameter** is defined.

**parameter-name**

Identifies a parameter of an SQL routine.
Usage

Referencing Routine Parameters

Routine parameters can only be referenced within the body of the SQL routine in which they are defined. A routine parameter is global to the SQL routine. It can be referenced anywhere in the body of the routine.

Avoiding Ambiguous References

The name of a routine parameter can be the same as the name of a local variable, a column, or another schema-defined entity such as a table. To avoid ambiguity when referencing these objects, qualification can be used as follows:

- A local variable can be qualified with the label of the compound statement in which it is declared.
- A routine parameter can be qualified with its associated schema and routine name.
- A column can be qualified with its schema and table name.
- Other schema-defined objects can be qualified with the name of the schema in which they are defined.

Resolving Ambiguous References

If a name is not qualified and more than one object has the specified name, CA IDMS uses the following precedence rules to resolve the ambiguous reference:

- If a local variable with a matching name has been declared within the compound statement in which the reference occurs, the reference is to the local variable. If more than one such variable is declared, the reference is to the variable declared in the innermost compound statement containing the reference.
- If a parameter of the routine in which the reference occurs has a matching name, the reference is to the routine parameter.
- Otherwise, the reference is treated as a reference to a schema-defined object. For information about how such a reference is resolved, see Resolving References to Entities in Schemas.

Note: In the SQL standard, an unqualified reference would be to the object with innermost scope.
Example

In the following SQL procedure, three routine parameters, TITLE, P_EMP_ID, and P_LAST_NAME are defined. The references to P_EMP_ID and P_LAST_NAME in the SELECT statement are qualified. The SET statement uses an unqualified reference to TITLE.

```
set options command delimiter '++';
create procedure SQLROUT.GETLNAME
  ( TITLE     varchar(10) with default
    , P_EMP_ID  NUMERIC(4)
    , P_LAST_NAME    varchar(25)
  )
  external name GETLNAME language SQL
L_MAIN: begin not atomic
  select EMP_FNAME
  into SQLROUT.GETLNAME.P_LAST_NAME
  from DEMOEMPL.EMPLOYEE
  where EMP_ID = SQLROUT.GETLNAME.P_EMP_ID;
  set TITLE = 'Success';
end L_MAIN ++
call SQLROUT.GETLNAME  ('?',2010)++ *
  *+ TITL:EP_EMP_ID   P_LAST_NAME
  *+ -----   ---------   ---------
  *+ Success         2010   Cora
```

Dynamic Parameters

A dynamic parameter is a value supplied during the execution of a dynamic SQL statement. It allows a statement, such as an UPDATE or INSERT statement, to be prepared once but executed multiple times with different input values for each execution. It also allows a SELECT statement to be prepared once but be used with different selection criteria to retrieve different rows.
Using Dynamic Parameters

You indicate the presence of a dynamic parameter by specifying a dynamic parameter marker within the text of the SQL statement being prepared. A dynamic parameter marker is the question mark ("?") symbol. It can be specified anywhere that an input host variable can be specified, except as noted below.

When executing an SQL statement that contains one or more dynamic parameter markers, you supply values to be substituted in place of the markers through the USING clause on the EXECUTE statement. If the prepared SQL statement is a SELECT, the substitution values are supplied through the USING clause on the OPEN statement.

Parameter Data Types

When a statement containing a dynamic parameter marker is prepared, CA IDMS infers the data type of the substitution value by examining the context where the dynamic parameter marker appears. You may use the DESCRIBE statement (or the DESCRIBE option on the PREPARE statement) to determine the assumptions that CA IDMS has made about the data types of the dynamic parameters.

The data types of the actual substitution values do not need to be the same as those assumed by CA IDMS. However, they must be compatible with respect to the assignment operator. That is, the value passed at the time the statement is executed must be capable of being assigned to a variable of the data type assumed by CA IDMS.

Note: For more information about the assignment operation, see Comparison, Assignment, Arithmetic, and Concatenation Operations. (see page 66)

The following table outlines how CA IDMS infers the data type of a dynamic parameter from the context in which it is used.

<table>
<thead>
<tr>
<th>Context</th>
<th>data type of dynamic parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date-time value expressions</td>
<td></td>
</tr>
<tr>
<td>? + date or date + ?</td>
<td>Date duration (DECIMAL(8,0))</td>
</tr>
<tr>
<td>? + time or time + ?</td>
<td>Time duration (DECIMAL(6,0))</td>
</tr>
<tr>
<td>? + timestamp or timestamp + ?</td>
<td>Time duration (DECIMAL(6,0))</td>
</tr>
<tr>
<td>date - ?</td>
<td>Date duration (DECIMAL(8,0))</td>
</tr>
<tr>
<td>time - ?</td>
<td>Time duration (DECIMAL(6,0))</td>
</tr>
<tr>
<td>timestamp - ?</td>
<td>Time duration (DECIMAL(6,0))</td>
</tr>
<tr>
<td>? - date</td>
<td>DATE</td>
</tr>
<tr>
<td>? - time</td>
<td>TIME</td>
</tr>
</tbody>
</table>
## Dynamic Parameters

### Chapter 4: Values and Value Expressions

#### Context
- `? + labeled duration` or `? - labeled duration`
  - Data type of dynamic parameter: Date if duration is day, month, year; time if duration is hour, minute, second

#### Other value expressions
- `? + ? DAY/MONTH/YEAR/HOUR/MINUTE/SECOND`
  - Data type: DECIMAL(31,6)
- `? | | v or v | | ?`
  - Data type: VARCHAR (256)

#### Scalar functions
- `CAST (?) AS data-type`
- `CHAR_LENGTH (?)`
- `CHARACTER_LENGTH (?)`
- `COALESCE (v,...?,....)`
  - Data type: VARCHAR (256)
- `CONCAT (?,?)`
- `CONVERT(?, data-type)`
- `FLOAT (?)`
- `HEX (?)`
- `IFNULL(v,?)`
  - Data type: VARCHAR (256)
- `INTEGER (?)`
- `LCASE(?)`
- `LEFT (?, ?)`
- `LENGTH (?)`
- `LOCATE (?, ?, ?)`
- `LOWER (?)`
- `LTRIM (?)`
- `POSITION (? IN ?)`
- `PROFILE (?)`
- `RTRIM (?)`
## Dynamic Parameters

### Context

<table>
<thead>
<tr>
<th>Context</th>
<th>data type of dynamic parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBSTR (?, ?, ?) or SUBSTRING (? FROM ? FOR ?)</td>
<td>First is VARCHAR (256); second and third are INTEGER</td>
</tr>
<tr>
<td>TRIM (?) FROM ?</td>
<td>Both are VARCHAR (256)</td>
</tr>
<tr>
<td>UCASE (?)</td>
<td>VARCHAR (256)</td>
</tr>
<tr>
<td>UPPER (?)</td>
<td>VARCHAR (256)</td>
</tr>
<tr>
<td>VALUE (v,...,?,...)</td>
<td>Same as v (The first entry in the list cannot be a dynamic parameter)</td>
</tr>
<tr>
<td>VARGRAPHIC (?)</td>
<td>VARCHAR (256)</td>
</tr>
</tbody>
</table>

### Predicates

| ? comparison-operator v or v comparison-operator ? | Same as v |
| ? LIKE ? ESCAPE ? | All are VARCHAR (256) |
| ? BETWEEN v1 AND v2 | Same as v1 |
| v BETWEEN ? AND ? | Both same as v |
| v1 IN (v2,...,?,...) | Same as v1 |
| ? IN (v1, v2, ...) | Same as v1 |
| ? comparison-operator ANY/ALL (subquery) | Same data type as result of subquery. |
| ? comparison-operator (subquery) | Same data type as result of subquery. |

### Update values

| UPDATE ... SET column = ? | Same as column. |
| INSERT ... VALUES (...,...) | Same as target column. |

**Note:** Dynamic parameters are always nullable.
Data Type Conversion Considerations

CA IDMS uses the rules in the previous table to infer a data type for a dynamic parameter. The actual value of the parameter may have a different data type provided the two are compatible with regard to the assignment operator. However, in certain cases, compatibility may not be sufficient. For example, if you wish to supply a very long character string as an input value and CA IDMS has inferred a data type of VARCHAR(256), the input value is truncated to a length of 256. To circumvent this, you can use the CAST function to override the default data type, as in the next example:

```
UPDATE MY.TEXT
SET STRING =
    CHAR(CURRENT TIMESTAMP) || '**' || CAST (?) AS VARCHAR(1000))
WHERE ...
```

As an operand of a concatenate symbol, CA IDMS would normally assign VARCHAR (256) as the data type for the dynamic parameter. However, by using a CAST function, the parameter is instead assigned a data type of VARCHAR (1000).

Restrictions in the Use of Dynamic Parameters

Dynamic parameter markers may not be used in the following contexts:

- Following a unary + or - operator
- By itself as an entry in the select-list of a query expression
- As both operands of a dyadic operator (except for the concatenation operator)
- As the first entry in the operand list of the COALESCE and VALUE functions
- As both the first and second or first and third operands of a BETWEEN predicate
- As both the first operand and any entry in the second operand of the IN predicate
- As the operand in the following functions: CHAR, DATE, DAY, DAYS, DECIMAL, DIGITS, MINUTE, MONTH, MICROSECOND, OCTET_LENGTH, SECOND, TIME, TIMESTAMP, YEAR

**Note:** The CAST function may be used to assign a data type to a parameter that otherwise would not be allowed within the desired context. For example, if you want to use a dynamic parameter as the first operand in the VALUE function, you may embed the parameter in a CAST function to assign a default data type.
Expansion of Dynamic-parameter-marker

Statement Options

For more information, refer to the options that make use of dynamic parameters on the following statements:

- DESCRIBE
- EXECUTE
- OPEN
- PREPARE

Expansion of Dynamic-parameter-marker

The expanded parameters of dynamic-parameter-marker indicate the use of a dynamic parameter.

Syntax

```
Expansion of dynamic-parameter-marker

?  ➤➤─────────────────────────────────────────────────────────────────────── ➤◄
```

Parameters

?  

Indicates that a dynamic parameter is used to supply a value when the statement is executed.

Usage

Dynamic SQL Only

Dynamic parameter markers may appear only within the text of an SQL statement which is compiled dynamically using the PREPARE statement. They may not be used in statements compiled through an EXECUTE IMMEDIATE statement nor in statements embedded in a host application program.

Note: For more information about the use of dynamic parameters, see Dynamic Parameters.
Example

The following INSERT statement contains dynamic parameter markers to indicate that values for the associated dynamic parameters are supplied when the statement is executed. The EXECUTE statement that follows, supplies those values through the use of host variables.

```
insert into coverage (plan_code, emp_id, selection_date, num_dependents)
values (?, ?, current date, ?)

execute dyninsert using :
wk-plan, :wk-emp, :wk-deps indicator :wk-deps-i
```

Special Registers

A special register is a system-supplied variable defined by CA IDMS. At any given time, the value of a special register depends upon the context of the current user session.

Usage

You use special registers in place of literals primarily in SQL data manipulation statements. For example, in the following SELECT statement, the special register CURRENT DATE specifies the end of a range of dates used as a selection criterion:

```
select emp_id, emp_lname
from employee
where start_date between '1989-01-01' and current date;
```

Expansion of Special-register

The expanded parameters of special-register identify system-supplied variables whose value is determined when the SQL statement in which they appear is executed.

Syntax

```
Expansion of special-register
```

```
USER
  GROUP
  CURRENT DATE
  CURRENT TIME
  CURRENT TIMESTAMP
  CURRENT TIMEZONE
  CURRENT DATABASE
  CURRENT SCHEMA
  CURRENT SQLID
```
**Note:** SQL Standard Compatibility. All special registers except USER are CA IDMS extensions of the SQL standard.

**Parameters**

**USER**
Contains the authorization identifier of the user executing the SQL session.
This value is established when the user signs on to the teleprocessing monitor or when the batch application is started by the operating system.
If no user has been established, the value of USER is blanks.

**GROUP**
Contains the default group identifier associated with the executing user as defined to the security facility.
If no user has been established or if the user has not been assigned a default group, the value of GROUP is blank.

**CURRENT DATE**
Contains the current date when the SQL statement is executed.

**CURRENT TIME**
Contains the current time when the SQL statement is executed.

**CURRENT TIMESTAMP**
Contains the current date and time when the SQL statement is executed with a precision of millionths of a second.

**CURRENT TIMEZONE**
Contains the difference between current time and Greenwich Mean Time expressed as a time duration.
This value is calculated from operating system values.

**CURRENT DATABASE**
Contains the name of the database to which the SQL session is connected.

**CURRENT SCHEMA**
Contains the current schema identifier associated with the SQL session.
This value is established in the CURRENT SCHEMA parameter of a SET SESSION statement. If the SET SESSION statement has not been issued, it is the value of the SCHEMA profile variable associated with the user session.
If no value has been established, the value of CURRENT SCHEMA is blanks.

**CURRENT SQLID**
Is a synonym for CURRENT SCHEMA.
## Usage

### Values in CURRENT DATE, TIME, and TIMESTAMP

All occurrences of CURRENT DATE, CURRENT TIME, and CURRENT TIMESTAMP appearing within a single SQL statement are effectively evaluated at the same time.

### Data Types and Equivalent Scalar Functions of Special Register Variables

This table gives the data types of CA IDMS special registers and the equivalent scalar function invocation:

<table>
<thead>
<tr>
<th>Special register</th>
<th>Data type</th>
<th>Equivalent Scalar Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER</td>
<td>CHARACTER(18)</td>
<td>USER()</td>
</tr>
<tr>
<td>GROUP</td>
<td>CHARACTER(18)</td>
<td></td>
</tr>
<tr>
<td>CURRENT DATE</td>
<td>DATE</td>
<td>CURDATE()</td>
</tr>
<tr>
<td>CURRENT TIME</td>
<td>TIME</td>
<td>CURTIME()</td>
</tr>
<tr>
<td>CURRENT TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>NOW()</td>
</tr>
<tr>
<td>CURRENT TIMEZONE</td>
<td>DECIMAL (6,0)</td>
<td></td>
</tr>
<tr>
<td>CURRENT SCHEMA</td>
<td>CHARACTER (18)</td>
<td></td>
</tr>
<tr>
<td>CURRENT SQLID</td>
<td>CHARACTER (18)</td>
<td></td>
</tr>
<tr>
<td>CURRENT DATABASE</td>
<td>CHARACTER (8)</td>
<td>DATABASE()</td>
</tr>
</tbody>
</table>

### ROWID Pseudo-column

A pseudo-column is a column automatically associated by CA IDMS with each table or view. Pseudo-columns are not part of the definition of a table or a view and thus do not show up as rows from the SYSTEM.COLUMN catalog table, nor will they be part of a SELECT * list.

ROWID has a special data type TID (Tuple ID) with a fixed length of 8 bytes. For any practical considerations, the TID data type can be considered equivalent to BIN(8). The ROWID contains information about the storage location of the row in the database. Internally the ROWID is made up of the DBKEY of the underlying database record (first 4 bytes). The last 4 bytes are currently ignored, but may be used in the future.

The value of ROWID is unique for each row of a base table; however, you cannot consider it to be a table's primary key because its value can change over the lifetime of the database. This could happen, for example, after an UNLOAD/RELOAD operation.
The ROWID provides unique access and the fastest access to a row of a table, no matter if the table is SQL- or non-SQL-defined.

The ROWID value is not persistent for the life of the database, but it never changes within a transaction or other controlled processes, if the row is not deleted, of course.

The value of ROWID can be null (for example, as the result of an outer join operation).

A ROWID pseudo-column cannot be updated or inserted.

Views also have an associated ROWID pseudo-column. The value of a view’s ROWID is the ROWID of the first base table in the decomposition of the view from left to right. The ROWID values of a view are not necessarily unique.

The ROWID pseudo-column is a CA IDMS extension of the SQL standard.

When to Use ROWID

Although ROWID can be used for SQL-defined tables, it is most useful for updating non-SQL-defined databases. Since such databases tend to have record types with no primary or foreign keys, identifying a specific row to be updated or deleted is often difficult. For such record types, it was often necessary to implement a table procedure to perform the update or deletion. The presence of ROWID pseudo-column makes the table procedure unnecessary, because it uniquely identifies each row of any non-SQL-defined table.

Expansion of rowid-pseudo-column

The expanded parameters of rowid-pseudo-column request the ROWID values to be determined when the SQL statement in which they appear is executed.

Syntax

Expansion of rowid-pseudo-column

\[
\text{ROWID} \quad \text{table-identifier.} \quad \text{schema-name.} \\
\text{alias.} \quad \text{view-identifier.}
\]
Parameters

*schema-name*

Specifies the schema with which the table or view identified by table-identifier or view-identifier is associated.

**Note:** For more information about using a schema name to qualify a table or view identifier, see Identifying Entities in Schemas.

*table-identifier*

Identifies a base table defined in the dictionary.

*view-identifier*

Identifies a view defined in the dictionary.

*alias*

Specifies the alias associated with the table or view to which the ROWID pseudo-column refers. The alias must be defined in the FROM parameter of the subquery, query specification, or SELECT statement that includes the ROWID.

Usage

Because the pseudo-column ROWID obviously becomes easily ambiguous when multiple tables or views are involved in an SQL statement, qualification is required in most, but the simplest statements.

**Note:** ROWID can generally also be used for tables associated with native VSAM files. However for KSDS native VSAM files ROWID cannot be used to directly access a KSDS record.
Examples

Using ROWID in a Simple SELECT Statement

```sql
SELECT ROWID, OFFICE_CODE_0450, OFFICE_CITY_0450
FROM EMPSCHM.OFFICE;
```

<table>
<thead>
<tr>
<th>ROWID</th>
<th>OFFICE_CODE_0450</th>
<th>OFFICE_CITY_0450</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'01259701'</td>
<td>002</td>
<td>BOSTON</td>
</tr>
<tr>
<td>X'0125A001'</td>
<td>001</td>
<td>SPRINGFIELD</td>
</tr>
<tr>
<td>X'0125A301'</td>
<td>005</td>
<td>GLASSER</td>
</tr>
<tr>
<td>X'0125A601'</td>
<td>012</td>
<td>CAMBRIDGE</td>
</tr>
<tr>
<td>X'0125A901'</td>
<td>008</td>
<td>WESTON</td>
</tr>
</tbody>
</table>

The values of ROWID are displayed as hexadecimal values, which in this case are also the values of the DBKEY for the OFFICE record in the non-SQL-defined schema EMPSCHM VERSION 100 of the demo employee database.

Using ROWID in the WHERE clause of a Searched UPDATE Statement

```sql
UPDATE EMPSCHM.EMPLOYEE SET EMP_CITY = 'BRUSSELS'
WHERE ROWID = X'0124FF01';
```

The column EMP_CITY of the EMPLOYEE record in the non-SQL schema EMPSCHM VERSION 100 is updated for the record whose DBKEY is X'0124FF01'.

Using ROWID in a JOIN of a Base Table and a View

Both examples use DEFJE01.EMPLOYEEV which is defined as follows:

```sql
CREATE VIEW DEFJE01.EMPLOYEEV
AS SELECT * FROM EMPSCHM.EMPLOYEE;
```

In the first example DEFJE01.EMPOFFV is defined as follows:

```sql
CREATE VIEW DEFJE01.EMPOFFV
AS SELECT EV.*, O.*
FROM EMPSCHM.OFFICE O, DEFJE01.EMPLOYEEV EV
WHERE "OFFICE-EMPLOYEE";
```
The returned ROWID for the view is the ROWID of the EMPSCHM.OFFICE base table:

```sql
SELECT EOV.ROWID, D.ROWID, D.*, EMP_ID, OFFICE_CODE_0450
FROM DEFJE01.EMPOFFV EOV, EMPSCHM.DEPARTMENT D
WHERE "DEPT-EMPLOYEE" AND EMP_ID < 5;
```

<table>
<thead>
<tr>
<th>ROWID</th>
<th>ROWID</th>
<th>DEPT_ID_0410</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'0125A001'</td>
<td>X'0125BD01'</td>
<td>100</td>
</tr>
<tr>
<td>X'0125A001'</td>
<td>X'0125BC01'</td>
<td>3100</td>
</tr>
<tr>
<td>X'0125A001'</td>
<td>X'0125AB01'</td>
<td>3200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPT_NAME_0410</th>
<th>DEPT_HEAD_ID_0410</th>
<th>EMP_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXECUTIVE ADMINISTRATION</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>INTERNAL SOFTWARE</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>COMPUTER OPERATIONS</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OFFICE_CODE_0450</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
</tr>
<tr>
<td>001</td>
</tr>
<tr>
<td>001</td>
</tr>
</tbody>
</table>
In the second example, DEFJE01.EMPOFFV is defined as follows:

```sql
CREATE VIEW DEFJE01.EMPOFFV
  AS SELECT EV.*, O.*
  FROM DEFJE01.EMPLOYEEV EV, EMPSCHM.OFFICE O
  WHERE "OFFICE-EMPLOYEE";
```

The returned ROWID for the view is the ROWID of the EMPSCHM.EMPLOYEE base table, which is the first base table in the view EMPLOYEEV.

```sql
SELECT EOV.ROWID, D.ROWID, D.*, EMP_ID, OFFICE_CODE_0450
  FROM DEFJE01.EMPOFFV EOV, EMPSCHM.DEPARTMENT D
  WHERE "DEPT-EMPLOYEE" AND EMP_ID < 5;
```

<table>
<thead>
<tr>
<th>ROWID</th>
<th>ROWID</th>
<th>DEPT_ID_0410</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'01252801'</td>
<td>X'0125BD01'</td>
<td>100</td>
</tr>
<tr>
<td>X'01253B01'</td>
<td>X'0125BC01'</td>
<td>3100</td>
</tr>
<tr>
<td>X'01255301'</td>
<td>X'0125AB01'</td>
<td>3200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPT_NAME_0410</th>
<th>DEPT_HEAD_ID_0410</th>
<th>EMP_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXECUTIVE ADMINISTRATION</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>INTERNAL SOFTWARE</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>COMPUTER OPERATIONS</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OFFICE_CODE_0450</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td></td>
</tr>
</tbody>
</table>

Expansion of rowid-pseudo-column
Searched Update of Records Without Primary Key

This example updates all the COVERAGE records of the employee with EMP_ID=23:

```
UPDATE EMPSCHM.COVERAGE C
    SET SELECTION_YEAR_0400 = 20
WHERE C.ROWID IN (
    SELECT CI.ROWID
    FROM EMPSCHM.EMPLOYEE E, EMPSCHM.COVERAGE CI
    WHERE "EMP-COVERAGE"
    AND EMP_ID = 23);
```  

Status = 0        SQLSTATE = 00000  
2 rows processed  

Searched Delete of Records Without Primary Key

This example deletes all the COVERAGE records of the employee with EMP_ID=23:

```
DELETE FROM EMPSCHM.COVERAGE C
WHERE C.ROWID IN (
    SELECT CI.ROWID
    FROM EMPSCHM.EMPLOYEE E, EMPSCHM.COVERAGE CI
    WHERE "EMP-COVERAGE"
    AND EMP_ID = 23);
```  

Status = 0        SQLSTATE = 00000  
2 rows processed  

Expansion of Value-expression

The expanded parameters of value-expression represent a single data value or a set of one or more data values in an SQL statement.
Expansion of Value-expression

Syntax

Expansion of value-expression

Parameters

+, -

Specifies the unary arithmetic operation to be performed on the operand that follows:

- + leaves the sign of the operand unchanged. A positive value remains positive. A negative value remains negative.
- - reverses the sign of the operand. A positive value becomes negative. A negative value becomes positive.

The default is +.

You can specify unary arithmetic operators with numeric operands only.

aggregate-function

Specifies an aggregate function to be used as an operand in the value expression. For expanded aggregate-function syntax, see Aggregate-function.

scalar-function

Specifies a scalar function to be used as an operand in the value expression. For expanded scalar-function syntax, see Expansion of Scalar-function.
column-name

Specifies a column to be used as an operand in the value expression. The expression is evaluated once for each value in the named column.

table-name.

Specifies the table, view, procedure or table procedure that includes the named column. For expanded table-name syntax, see Expansion of Table-name.

alias.

Specifies the alias associated with the table, view, procedure or table procedure that includes the named column. The alias must be defined in the FROM parameter of the subquery, query specification, or SELECT statement that includes the value expression.

literal

Specifies a literal to be used as a single operand in the value expression. For expanded literal syntax, see Expansion of Literal.

host-variable

Specifies a host variable to be used as a single operand in the value expression. For expanded host-variable syntax, see Expansion of Host-variable.

special-register

Specifies a special register to be used as a single operand in the value expression. For expanded special-register syntax, see Expansion of Special-register.

(value-expression)

Specifies another value expression to be used as a single operand in the value expression. To be manipulated as a single operand, the value expression must be enclosed in parentheses.

labeled-duration

Specifies a labeled duration to be used as an operand in the value expression. For expanded labeled-duration syntax, see Expansion of Labeled-duration.

dynamic-parameter-marker

Specifies a dynamic parameter to be used as a single operand in the value expression. For expanded dynamic-parameter-marker syntax, see Expansion of Dynamic-parameter-marker.

rowid-pseudo-column

Requests the ROWID value to be determined when the SQL statement in which it appears is executed.
**routine-parameter**

Specifies a routine parameter to be used as a single operand in the value expression.

**Note:** For more information about expanded `routine-parameter` syntax, see Expansion of Routine-parameter.

**local-variable**

Specifies a local variable to be used as a single operand in the value expression.

**Note:** For more information about expanded `local-variable`, see Local Variables. (see page 81)

*, /, +, -, ||

Specifies the binary arithmetic operation or concatenation operation to be performed on the operands preceding and following the operator.

Binary arithmetic operators are:

- * multiplies the first operand by the second operand
- / divides the first operand by the second operand
- + adds the second operand to the first operand
- - subtracts the second operand from the first operand

You can specify binary arithmetic operators with numeric operands only.

The concatenation operator is:

- || concatenates the second operand to the first operand

You can specify the concatenation operator with binary operands, character operands, or graphics operands.
Usage

Order of Evaluation

After evaluating the individual operands, CA IDMS performs the operations in a value expression in the following order:

1. Unary operations from left to right.
2. Multiplication and division from left to right.
3. Addition and subtraction from left to right.

You can use parentheses to override the default order of evaluation. Operations in parentheses are performed first.

For example, the result of the following value expression is 19:

\[ 10 \times 2 - 1 \]

When the subtraction operation is enclosed in parentheses, the result of the expression is 10:

\[ 10 \times (2 - 1) \]

Unary Operators With Signed Numeric Literals

If the operand following a unary operator is a numeric literal that includes a plus or minus sign, the literal must be enclosed in parentheses.

Null Values in a Value Expression

If the value of any of the operands in a value expression is null, the result of the expression is a null value.

Data Type of the Result

The data type of the result of a value expression with one operand is the data type of the operand.

The data type of the result of a numeric value expression with multiple operands is the common data type corresponding to the data type of highest precedence in the expression, as determined by the rules for data type conversion in arithmetic operations.
This table shows the data type of the result of a concatenation operation for each allowable combination of operands:

<table>
<thead>
<tr>
<th>Operand</th>
<th>Operand</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>CHARACTER</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>VARCHAR</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>VARCHAR</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>BINARY</td>
<td>BINARY</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>BINARY</td>
<td>CHARACTER</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>BINARY</td>
<td>VARCHAR</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>GRAPHIC</td>
<td>GRAPHIC</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>VARGRAPHIC</td>
<td>GRAPHIC</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>VARGRAPHIC</td>
<td>VARGRAPHIC</td>
</tr>
</tbody>
</table>

**Note:** For more information about data type conversion, see Comparison, Assignment, Arithmetic, and Concatenation Operations.

**Examples**

**A Single Operand**

In the SELECT statement below, the value expressions that identify the data to be selected each consist of a single operand. The first is a column, and the second two are aggregate functions.

```sql
select proj_leader_id, count(proj_id), avg(est_man_hours) from project group by proj_leader_id;
```

**Multiple Operands**

In the UPDATE statement below, the value expression that specifies the new value for SALARY_AMOUNT includes multiple operands. CA IDMS computes the new value by multiplying the value in the SALARY_AMOUNT column by .06, adding the result to the original value in SALARY_AMOUNT, and then adding the value in :MERIT_AMT to the result of the first addition.

```sql
EXEC SQL UPDATE POSITION
  SET SALARY_AMOUNT = SALARY_AMOUNT + (SALARY_AMOUNT * .06) + :MERIT_AMT
  WHERE EMP_ID = :EMPLOYEE-ID
END-EXEC
```
Durations

A duration is a value that represents a time interval. There are three kinds of duration:

- Labeled duration
- Date duration
- Time duration

Labeled Durations

A labeled duration represents a specific unit of time as expressed by a value expression followed by a duration keyword. Labeled duration appears in syntax as labeled-duration. For expanded labeled-duration syntax, see Expansion of Labeled-duration.

These are examples of labeled durations:

- 10 MINUTES
- :HV-INPUT YEARS
- SUM (HOURS_WORKED) HOURS

Date Duration

A date duration is a value of data type DECIMAL(8,0) that represents an interval of years, months, and days. The value must have the format yyyyymmdd where yyyy represents the number of years, mm the number of months, and dd the number of days.

For example, the date duration 41027 represents 4 years, 10 months, and 27 days.

Time Duration

A time duration is a value of data type DECIMAL(6,0) that represents an interval of hours, minutes, and seconds. The value must have the format hhmmss where hh represents the number of hours, mm the number of minutes, and ss the number of seconds.

For example, the time duration 173306 represents 17 hours, 33 minutes, and 6 seconds.

Expansion of Labeled-duration

The expanded parameters of labeled-duration specify an interval of time in a unit of measure ranging from microseconds to years.
Expansion of Labeled-duration

Syntax

Expansion of labeled-duration

```
aggregate-function | column-name
| table-name. | alias. | literal | host-variable | value-expression | dynamic-parameter-marker
```

Parameters

**aggregate-function**

Specifies the aggregate function that represents the value in the labeled duration. For expanded *aggregate-function* syntax, see Aggregate-function.

**column-name**

Specifies the column that represents the value in the labeled duration.

**table-name.**

Specifies the table, view, procedure or table procedure that includes the named column. For expanded *table-name* syntax, see Expansion of Table-name.

**alias.**

Specifies the alias for the table, view, procedure or table procedure that includes the named column.

**literal**

Specifies the literal that represents the value in the labeled duration. For expanded *literal* syntax, see Expansion of Literal.

**host-variable**

Specifies the host variable that contains the value in the labeled duration. For expanded *host-variable* syntax, see Expansion of Host-variable.

**value-expression**

Specifies the value expression that represents the value in the labeled duration. For expanded *value-expression* syntax, see Expansion of Value-expression.
**dynamic-parameter-marker**

Specifies that the value in the labeled-duration statement is supplied as a dynamic parameter. For expanded **dynamic-parameter-marker** syntax, see Expansion of Dynamic-parameter-marker.

**YEAR / YEARS**

Indicates that the unit of measure of the duration is years.

**MONTH / MONTHS**

Indicates that the unit of measure of the duration is months.

**DAY / DAYS**

Indicates that the unit of measure of the duration is days.

**HOUR / HOURS**

Indicates that the unit of measure of the duration is hours.

**MINUTE / MINUTES**

Indicates that the unit of measure of the duration is minutes.

**SECOND / SECONDS**

Indicates that the unit of measure of the duration is seconds.

**MICROSECOND / MICROSECONDS**

Indicates that the unit of measure of the duration is microseconds.
Date/time Arithmetic

The only arithmetic operations that can be performed on date/time values are addition and subtraction.

Date/time Addition

If a date/time value is the operand of addition, the other operand must be a duration. These rules govern the use of the addition operator with date/time values:

- If one operand is a date, the other operand must be a date duration or labeled duration of years, months, or days
- If one operand is a time, the other operand must be a time duration or a labeled duration of hours, minutes, or seconds
- If one operand is a timestamp, the other operand must be a duration

Date/time Subtraction

The rules for the use of the subtraction operator on date/time values differ from those for addition. The tables below describe the rules for using the subtraction operator with date/time values.

Note: The second operand cannot be a timestamp.

First Operand Rules

<table>
<thead>
<tr>
<th>If the first operand is</th>
<th>The second operand must be</th>
</tr>
</thead>
<tbody>
<tr>
<td>A date</td>
<td>■ A date</td>
</tr>
<tr>
<td></td>
<td>■ A date duration</td>
</tr>
<tr>
<td></td>
<td>■ A string representation of a date</td>
</tr>
<tr>
<td></td>
<td>■ A labeled duration of years, months or days</td>
</tr>
<tr>
<td>A time</td>
<td>■ A time</td>
</tr>
<tr>
<td></td>
<td>■ A time duration</td>
</tr>
<tr>
<td></td>
<td>■ A string representation of a time</td>
</tr>
<tr>
<td></td>
<td>■ A labeled duration of hours, minutes, or seconds</td>
</tr>
<tr>
<td>A timestamp</td>
<td>■ A duration</td>
</tr>
</tbody>
</table>
### Second Operand Rules

<table>
<thead>
<tr>
<th>If the second operand is</th>
<th>The first operand must be</th>
</tr>
</thead>
<tbody>
<tr>
<td>A date</td>
<td>■ A date</td>
</tr>
<tr>
<td></td>
<td>■ A string representation of a date</td>
</tr>
<tr>
<td>A time</td>
<td>■ A time</td>
</tr>
<tr>
<td></td>
<td>■ A string representation of a time</td>
</tr>
</tbody>
</table>

### Date Arithmetic

Dates can be subtracted, incremented, or decremented.

**Subtracting Dates**

The result of subtracting one date from another date is a date duration in the form `yyyyymmdd` that specifies the number of years, months, and days between the two dates. The data type of the result is `DECIMAL(8,0)`.

In the expression `D1 - D2`, where `D1` and `D2` are date values:

**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 made negative
Date/Time Arithmetic

Date Subtraction Procedures

These are the procedures used to obtain a result R in the expression \( R = D_1 - D_2 \) where \( D_1 \) and \( D_2 \) are date values:

If \( \text{DAY}(D_2) \leq \text{DAY}(D_1) \),
then \( \text{DAY}(R) = \text{DAY}(D_1) - \text{DAY}(D_2) \).

If \( \text{DAY}(D_2) > \text{DAY}(D_1) \),
then \( \text{DAY}(R) = N + \text{DAY}(D_1) - \text{DAY}(D_2) \)
where \( N \) = the last day of \( \text{MONTH}(D_2) \).
\( \text{MONTH}(D_2) \) is then incremented by 1.

If \( \text{MONTH}(D_2) \leq \text{MONTH}(D_1) \),
then \( \text{MONTH}(R) = \text{MONTH}(D_1) - \text{MONTH}(D_2) \).

If \( \text{MONTH}(D_2) > \text{MONTH}(D_1) \),
then \( \text{MONTH}(R) = 12 + \text{MONTH}(D_1) - \text{MONTH}(D_2) \).
\( \text{YEAR}(D_2) \) is then incremented by 1.

\( \text{YEAR}(R) = \text{YEAR}(D_1) - \text{YEAR}(D_2) \).

Example of Subtracting Dates

The result of \( \text{DATE}('12/31/2000') - \text{DATE}('8/10/1999') \) is 10421, representing a duration of 1 year, 4 months, and 21 days.

Arithmetic with a Date and a Duration

What You Can Do

The result of adding a duration to a date or subtracting a duration from a date is a date. The result must fall in the range of dates from January 1, 0001 to December 31, 9999.

:warning. If an invalid date is calculated during an UPDATE STATEMENT, the target column remains unchanged. An invalid date can be calculated during "Operations with a Duration of Years" or "Operations with a Duration or Months."

Operations with a Duration of Years

Adding or subtracting a duration of years affects the year of the resulting date but does not affect the month or day unless the result is February 29 of a non-leap year. In this case, the day portion is set to 28. When this adjustment is required, a warning message is issued.

For example, the result of \( \text{DATE}('5/1/1998') + 3 \) YEARS is '5/1/2001'.

112 SQL Reference Guide
Operations with a Duration of Months

Adding or subtracting a duration of months affects the month and potentially the year of the resulting date. The day portion of the date is unchanged unless the result is an invalid date, such as June 31. When an invalid date is calculated, CA IDMS returns a warning message. If the invalid date is calculated during a SELECT statement, the date is set to the last day of the month.

For example, the result of DATE('10/31/2001') - 1 MONTH is '9/30/2001'.

Operations with a Duration of Days

Adding or subtracting a duration of days affects the day of the resulting date and potentially the month and year.

For example, the result of DATE('12/15/2000') + 45 DAYS is '1/29/2001'.

Operations with Date Durations

Date durations of data type DECIMAL (8,0) in the form yyyyymmdd may also be added to and subtracted from dates. The date duration may be a positive or negative value. The result is a date that has been incremented or decremented by the specified number of years, months, and days, respectively. Thus, D1 + N, where N is a positive date duration, is equivalent to this expression:

D1 + YEAR(N) YEARS + MONTH(N) MONTHS + DAY(N) DAYS

For example, the result of DATE('4/13/2001') + 101 is '5/14/2001'.

Note: Leading zeros are dropped. Therefore, 101 is the same as 00000101.

Reversing Operations with Date Durations

If you add duration 100 (one month) to date D1, obtaining result R, R - 100 may not necessarily equal D1 because the operation D1 + 100 may require an end-of-the-month adjustment. For example:

DATE('8/31/2001') + 100 = '9/30/2001'

However:

DATE('9/30/2001') - 100 = '8/30/2001'
Time Arithmetic

Times can be subtracted, incremented, or decremented.

Subtracting Times

The result of subtracting one time (T2) from another (T1) is a time duration in the form $hh:mm:ss$ that specifies the number of hours, minutes, and seconds between the two times. The data type of the result is DECIMAL (6,0).

In the expression $T1 - T2$, where $T1$ and $T2$ are time values:

If $T1$ is greater than or equal to $T2$,

$T2$ is subtracted from $T1$

If $T1$ is less than $T2$,

$T1$ is subtracted from $T2$, and the sign of the result is made negative

Time Subtraction Procedures

These are the procedures used to obtain a result $R$ in the expression $R = T1 - T2$ where $T1$ and $T2$ are time values:

If $SECOND(T2) \leq SECOND(T1)$,

then $SECOND(R) = SECOND(T1) - SECOND(T2)$.

If $SECOND(T2) > SECOND(T1)$,

then $SECOND(R) = 60 + SECOND(T1) - SECOND(T2)$.

MINUTE($T2$) is then incremented by 1.

If $MINUTE(T2) \leq MINUTE(T1)$,

then $MINUTE(R) = MINUTE(T1) - MINUTE(T2)$.

If $MINUTE(T2) > MINUTE(T1)$,

then $MINUTE(R) = 60 + MINUTE(T1) - MINUTE(T2)$.

HOUR($T2$) is then incremented by 1.

$HOUR(R) = HOUR(T1) - HOUR(T2)$.

For example, the result of TIME ('16:43:17') - TIME('14:30:00') is 21317, representing a duration of 2 hours, 13 minutes and 17 seconds.
Arithmetic with a Duration and a Time

What is the Result?

The result of adding a duration to a time, or of subtracting a duration from a time, is a time.

Operations With a Duration of Hours

Adding or subtracting a duration of hours affects the hours of the resulting time. The minutes and seconds are unchanged.

For example, the result of TIME ('16:43:17') + 3 HOURS is '19:43:17'.

Operations With a Duration of Minutes

Adding or subtracting a duration of minutes affects the minutes and potentially the hours of the resulting time. The second's portion of the time is unchanged.

For example, the result of TIME ('16:43:17') + 30 MINUTES is '17:13:17'.

Operations With a Duration of Seconds

Adding or subtracting a duration of seconds affects the seconds and potentially the minutes and hours of the resulting time.

For example, the result of TIME ('16:43:17') + 51 SECONDS is '16:44:08'.

Note: In arithmetic with a time and a duration, overflow or underflow of hours is discarded.

Operations With Time Durations

Time durations of a data type DECIMAL(6,0) may also be added to and subtracted from times. The time duration may be a positive or negative value.

The result is a time that has been incremented or decremented by the specified number of hours, minutes, and seconds, respectively. Thus, T1 + N, where N is a positive time duration, is equivalent to this expression:

T1 + HOUR(N) HOURS + MINUTE(N) MINUTES + SECONDS(N) SECONDS

For example, the result of TIME ('16:43:17') + 32114 is '20:08:31'.


Timestamp Arithmetic

Timestamps can be incremented, or decremented. The result of adding a duration to a
timestamp, or of subtracting a duration from a timestamp, is a timestamp.

Timestamp arithmetic is performed as described for date and time arithmetic except
that an overflow or underflow of hours is carried into the date part of the result.

Precedence of Operations

Expressions within parentheses are evaluated first. When the order of evaluation is not
specified by parentheses:
■ Prefix operators are applied before multiplication and division
■ Multiplication and division are applied before addition and subtraction
■ Operators at the same precedence level are applied from left to right

Expansion of XML-value-expression

The expanded parameters of XML-value-expression specify XML values.

Note: For more information about XML values, see XML Data Type and XML Values.

Syntax

```
Expansion of XML-value-expression
   ┌─────────────────────────────►◄
   │ XML-value-function           |
   │ subquery                    |
```

Parameters

XML-value-function

Specifies an XML-value-function that returns an XML value. See Expansion of
XML-value-function, for more information.

subquery

Specifies a subquery that must return a single XML value or the NULL value. See
Subqueries, for more information.
Chapter 5: Functions

This section contains the following topics:

- **Aggregate-function** (see page 117)
- **Scalar Function** (see page 121)
- **CA IDMS Scalar Functions** (see page 124)
- **Expansion of User-defined-function** (see page 179)
- **Expansion of XML-value-function** (see page 181)
- **XML Value Functions** (see page 181)

**Aggregate-function**

An aggregate function is a function whose argument includes one or more columns and which operates on one or more rows. The result of an aggregate function is a single value. This value is derived from the sets of values in the columns named in the argument.

The set of values is derived from the result table or from each group of results if the associated query contains a GROUP BY.

**Expansion of Aggregate-function**

The expanded parameters of aggregate-function represent an aggregate function in an SQL statement.

**Syntax**

```
Expansion of aggregate-function

<table>
<thead>
<tr>
<th>AVG</th>
<th>MAX</th>
<th>MIN</th>
<th>SUM</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td></td>
<td></td>
<td></td>
<td>(</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>)</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>table-name</td>
<td>alias.</td>
<td>column-name</td>
<td></td>
</tr>
<tr>
<td>all value-expression</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DISTINCT</td>
<td>table-name</td>
<td>alias.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Parameters

AVG

Computes the arithmetic mean of the non-null values specified by the argument.

If no rows are found for the function, or if all rows found contain null values, the result of the function is null.

MAX

Finds the largest of the non-null values specified by the argument.

If no rows are found for the function, or if all rows found contain null values, the result of the function is null.

MIN

Finds the smallest of the non-null values specified by the argument.

If no rows are found for the function, or if all rows found contain null values, the result of the function is null.

SUM

Computes the total of the non-null values specified by the argument.

If no rows are found for the function, or if all rows found contain null values, the result of the function is null.

DISTINCT

Directs CA IDMS to exclude both duplicate values and null values from the set of values identified by column-name before evaluating the function.

If you do not specify DISTINCT, CA IDMS excludes only null values.

column-name

Specifies the set of values in the named column. CA IDMS excludes null values from the set before evaluating the function.

For AVG and SUM, the named column must have an approximate or exact numeric data type.

table-name

Specifies the table, view, procedure or table procedure that includes the named column. For expanded table-name syntax, see Expansion of Table-name.

alias

Specifies the alias associated with the table, view, procedure or table procedure that includes the named column. The alias must be defined in the FROM parameter of the subquery, query specification, or SELECT statement that includes the aggregate function.
all value-expression

Specifies the set of values derived from the evaluation of a value expression. Null values are excluded from the set before the function is evaluated.

When used as the argument of an aggregate function, value-expression:

- Must include at least one column reference
- Cannot include another aggregate function (that is, you cannot nest aggregate functions)
- Cannot include both arithmetic operators and outer references

   Note: For more information about outer references, see Subqueries. (see page 238)

For AVG and SUM, the values in the set must have an approximate or exact numeric data type.

The keyword ALL is optional and does not affect the evaluation of the function. For expanded value-expression syntax, see Expansion of Value-expression.

COUNT

Counts the number of rows where the column identified by the argument contains non-null values.

If no rows are found for the function, or if all rows found contain null values, the result of the function is 0.

* 

Counts the rows in the requested grouping, if any, of the result table.

Usage

The DISTINCT and ALL Parameters with MAX and MIN

When the argument of the MAX or MIN function is a single column, the result of the function is the same whether you use the DISTINCT parameter. However, CA IDMS evaluates the function more efficiently when you omit DISTINCT.

Aggregate Functions with Grouped Tables

When used in a subquery, query-specification, or SELECT statement that includes the GROUP BY parameter, aggregate functions are evaluated once for each group in the table.

If there is no GROUP BY parameter, aggregate functions are evaluated once for the entire result table.
Aggregate-function

Data Types of Function Results

<table>
<thead>
<tr>
<th>Function</th>
<th>Data type of the value returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>Determined by the rules for data type conversion in arithmetic operations</td>
</tr>
<tr>
<td>COUNT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MAX</td>
<td>Same as the data type of the values specified by the argument</td>
</tr>
<tr>
<td>MIN</td>
<td>Same as the data type of the values specified by the argument</td>
</tr>
<tr>
<td>SUM</td>
<td>Determined by the rules for data type conversion in arithmetic operations</td>
</tr>
</tbody>
</table>

Note: For more information about data type conversion, see Comparison, Assignment, Arithmetic, and Concatenation Operations. (see page 66)

Examples

Finding an Average

The following SELECT statement returns the average number of days employees took as vacation time or sick time in the 1989 fiscal year:

```
select avg(vac_taken + sick_taken)
  from benefits
  where fiscal_year = '89';
```

Counting Rows that Satisfy a Condition

The following SELECT statement counts the insurance plans not currently used by any employees:

```
select count(*)
  from insurance_plan
  where plan_code not in
    (select plan_code
     from coverage);
```
Counting Rows in Table Groupings

The following SELECT statement counts the number of different jobs in each department:

```sql
select d.dept_id, d.dept_name, count(distinct p.job_id)
from department d, employee e, position p
where d.dept_id = e.dept_id
    and e.emp_id = p.emp_id
group by d.dept_id, d.dept_name;
```

Selecting the Largest Value

The following SELECT statement identifies the jobs that have the largest number of positions open:

```sql
select job_id, job_title, num_open
from job
where num_open =
    (select max(num_open)
     from job);
```

More Information

- For more information about subqueries and query specifications, see Query Specifications, Subqueries, Query Expressions, and Cursor Specifications.
- For more information about the SELECT statement, see SELECT.

More Information:

- Query Specifications, Subqueries, Query Expressions, and Cursor Specifications (see page 231)
- SELECT (see page 518)

Scalar Function

A scalar function is a function that operates on 0 or more value expressions and returns a single value. This value is derived from the expression or expressions named in the function arguments.

Scalar functions can be user-defined or built-in. User-defined functions are defined using a CREATE FUNCTION statement. Built-in functions are known to the DBMS but are not explicitly defined. All built-in functions are provided as part of CA IDMS SQL. During the installation of CA IDMS, a large number of generally useful user-defined functions is defined into the SYSCA schema. For detailed descriptions of the CA-supplied scalar functions, see CA IDMS Scalar Functions.
### Expansion of Scalar-function

A scalar function operates on 0 or more value expressions, resulting in a single value.

### Syntax

```
user-defined-function
  ABS-function
  ACOS-function
  ASIN-function
  ATAN-function
  ATAN2-function
  CAST-function
  CEIL-function
  CEILING-function
  CHAR-function
  CHAR_LENGTH-function
  CHARACTER_LENGTH-function
  COALESCE-function
  CONCAT-function
  CONVERT-function
  COS-function
  COSH-function
  COT-function
  CURDATE-function
  CURTIME-function
  DATABASE-function
  DATE-function
  DAY-function
  DAYNAME-function
  DAYOFMONTH-function
  DAYOFYEAR-function
  DAYS-function
  DECIMAL-function
  DEGREES-function
  DIGITS-function
  EXP-function
  FLOAT-function
  FLOOR-function
  HEX-function
  HOUR-function
  IFNULL-function
  INSERT-function
  INTEGER-function
  LCASE-function
  LEFT-function
  LENGTH-function
  LOCATE-function
  LOG-function
  LOG10-function
  LOWER-function
  LTRIM-function
  MICROSECOND-function
  MINUTE-function
  MOD-function
  MONTH-function
  MONTHNAME-function
  NOW-function
  OCTET_LENGTH-function
  PI-function
  POSITION-function
  POWER-function
  PROFILE-function
  QUARTER-function
```
Parameters

user-defined-function

Specifies to invoke a user-defined function. See Expansion of User-defined-function, for more information about expanded user-defined-function syntax.

The remaining parameters are used to invoke a CA IDMS scalar function.

Note: For more information about the CA-supplied scalar functions, see CA IDMS Scalar Functions (see page 124).

Usage

Built-in Versus User-Defined Functions

The scalar functions that are provided by CA IDMS are implemented as built-in functions or user-defined functions. Built-in functions are not defined in the dictionary. The user-defined functions provided by CA IDMS are defined in the SYSCA schema during installation.
How a function is implemented is significant for two reasons:

- User-defined functions cannot be referenced in a table’s check constraint
- The number of user-defined functions that can be referenced in an SQL statement is limited

Note: For more information about how to determine the implementation of a CA IDMS supplied scalar function, see CA IDMS Scalar Functions (see page 124).

More information:

CA IDMS Scalar Functions (see page 124)

CA IDMS Scalar Functions

This section describes the scalar functions provided by CA IDMS including their purpose, syntax, parameters, usage considerations, and examples.

ABS-function

Syntax

\[
\text{ABS}(\text{value-expression})
\]

ABS returns the absolute value of the value-expression, which must have a numeric data type.

The result has the same data type as the value-expression. If the value-expression is null, the result is the null value. If a data error occurs, an exception is raised.

Example The following statement returns 125:

```sql
SELECT ABS(-125)
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

ACOS-function

Syntax

\[
\text{ACOS}(\text{value-expression})
\]

ACOS returns the inverse cosine of the value-expression, which must have a numeric data type.

The result has the same data type as the value-expression. If the value-expression is null, the result is the null value. If a data error occurs, an exception is raised.
ACOS returns the arccosine of the value-expression as an angle expressed in radians. ACOS is the inverse function of the COS function.

The **value-expression** must be of any numeric data type and must have a value in the range of -1 to 1. It is converted to a double precision floating-point number for processing by this function.

The result of the function is a double precision floating-point number. If the value-expression is null, the result is the null value. If a data error occurs, an exception is raised.

**Example**

The following statement returns 7.9539883018414370E-01:

```sql
SELECT ACOS(0.7)
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**ASIN-function**

**Syntax**

```
ASIN ( value-expression )
```

ASIN returns the arcsine of the value-expression as an angle expressed in radians. ASIN is the inverse function of the SIN function.

The **value-expression** must be of any numeric type and must have a value in the range of -1 to 1. It is converted to a double precision floating-point number for processing by this function.

The result of the function is a double precision floating-point number. If the value-expression is null, the result is the null value. If a data error occurs, an exception is raised.

**Example**

The following statement returns 1.5707963267948966E+00:

```sql
SELECT ASIN(1)
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**ATAN-function**

**Syntax**

```
ATAN ( value-expression )
```

ATAN returns the arctangent of the value-expression as an angle expressed in radians. ATAN is the inverse function of the TAN function.

The **value-expression** must be of any numeric type and must have a value in the range of -infinity to infinity. It is converted to a double precision floating-point number for processing by this function.

The result of the function is a double precision floating-point number. If the value-expression is null, the result is the null value. If a data error occurs, an exception is raised.

**Example**

The following statement returns -1.5707963267948966E+00:

```sql
SELECT ATAN(-1)
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```
ATAN returns the arctangent of the value-expression as an angle expressed in radians.
ATAN is the inverse function of the TAN function.

The value-expression must be of any numeric data type. It is converted to a double precision floating-point number for processing by this function.

The result of the function is a double precision floating-point number. If the value-expression is null, the result is the null value. If a data error occurs, an exception is raised.

Example: The following statement returns 1.2490457723982544E+00

```
SELECT ATAN(3)
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**ATAN2-function**

**Syntax**

```
ATAN2 ( value-expression1, value-expression2 )
```

**Parameters**

value-expression1

Specifies a numeric value-expression. See Expansion of Value-expression.

value-expression2

Specifies a numeric value-expression. See Expansion of Value-expression.

ATAN2 returns the arctangent of x and y coordinates, given by value-expression1 and value-expression2 respectively, as an angle expressed in radians.

Both value-expressions must be of any numeric data type and cannot both be 0. They are converted to double precision floating-point numbers for processing by this function.

The result of the function is a double precision floating-point number. If the value-expression is null, the result is the null value. If a data error occurs, an exception is raised.

Example: The following statement returns 1.2490457723982544E+00

```
SELECT ATAN2(1,3)
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**CAST-function**

**Syntax**

```
CAST ( value-expression AS data-type )
```

```
NULL
```
The CAST function forces conversion of the value-expression to a specified data type.

CAST allows:
- All conversions that can be made without the CAST function
- A numeric value to character value conversion, which creates a display version of the numeric data type and follows truncation rules for the numeric data type

When an approximate numeric value is cast to a character value, the value is converted to an external floating point representation. Trailing zeros removed from the mantissa except in the first place to the right of the decimal point.

The maximum number of digits in the mantissa is 6 for a REAL value and 13 for a DOUBLE PRECISION value. The exponent is an integer value with at least one digit. A negative value in either the mantissa or the exponent is preceded by a sign character.

These are examples of casting REAL data values to character values:

<table>
<thead>
<tr>
<th>REAL data value</th>
<th>Character value through CAST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0098999E+02</td>
<td>1.009899E2</td>
</tr>
<tr>
<td>1.9899997E+00</td>
<td>1.989999E0</td>
</tr>
<tr>
<td>0.0000000E+00</td>
<td>0.0E0</td>
</tr>
<tr>
<td>9.9999964E-02</td>
<td>9.999964E-2</td>
</tr>
</tbody>
</table>

When an exact numeric value is cast to a character value, the numeric value is left-justified, with leading zeros removed except in the first place to the left of the decimal point and trailing zeros removed except in the first place to the right of the decimal point. Negative values are preceded by a sign character.

These are examples of casting exact numeric values:

<table>
<thead>
<tr>
<th>Exact numeric value</th>
<th>Character value through CAST</th>
</tr>
</thead>
<tbody>
<tr>
<td>001234.56</td>
<td>1234.56</td>
</tr>
<tr>
<td>00.123456</td>
<td>0.123456</td>
</tr>
<tr>
<td>-6.7000</td>
<td>-6.7</td>
</tr>
<tr>
<td>666.0000</td>
<td>666.0</td>
</tr>
<tr>
<td>666</td>
<td>666</td>
</tr>
</tbody>
</table>
A character value to numeric value conversion extracts the numeric value from the string in either decimal or floating point notation. The character value can have leading or trailing blanks but cannot have extraneous characters (for example, more than one sign or more than one decimal point).

A character value to graphics value conversion converts the character string to its DBCS equivalent and truncates or pads the result to conform to the length in the data type specification.

Note: For more information about assignment rules in conversions, see Comparison, Assignment, Arithmetic, and Concatenation Operations. (see page 66)

Parameters

NULL

Forces conversion of a null value to a specified data type.

AS data-type

Identifies the data type to which the value-expression or null value is to be converted. Expansion of data-type is presented under Expansion of Data-type.

CEIL or CEILING-function

Syntax

CEIL(value-expression)  
CEILING(value-expression)

CEILING returns the smallest integer value that is greater than or equal to the value-expression. CEIL and CEILING are identical.

The value-expression must be of any numeric data type.

The result of the function has the same data type as the value-expression except that the scale is 0 if the value-expression is of type (UNSIGNED) DECIMAL or (UNSIGNED) NUMERIC. For example, a value-expression with a data type of NUMERIC(3,2) results in NUMERIC(3,0). If the value-expression is null, the result is the null value. If a data error occurs, an exception is raised.

Example

The following statement returns: 13  2.0000000000000000E+00    -12

SELECT CEILING(12.55), CEILING(123.1E-2), CEILING (-12.55)  
   FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';

128 SQL Reference Guide
CHAR-function

Syntax

```
CHAR ( value-expression )
```

CHAR obtains a character string representation from the value in value-expression. The syntax and semantics for the CHAR function depends on the data type of value-expression.

- Data type of value-expression is an exact numeric data: INTEGER, SMALLINT, OR LONGINT.
- CHAR returns a fixed-length character string representation of the exact numeric value of value-expression. Specifying a second parameter is not allowed. The result is left-justified and contains \( n \) characters corresponding to the digits of the value of value-expression with a preceding minus sign if the value-expression is negative. The length of the returned string depends on the data type of value-expression. A SMALLINT data type has a result length of 6. An INTEGER has a result length of 11 and a LONGINT has a result length of 20.

Example:

```
SELECT CHAR(FIXLENGTH), LENGTH(CHAR(FIXLENGTH)) AS LEN_SMALLINT
     , CHAR(NUMROWS), LENGTH(CHAR(NUMROWS)) AS LEN_INTEGER
FROM SYSTEM.TABLE WHERE NAME = 'TABLE';
```

<table>
<thead>
<tr>
<th>CHAR(FUNCTION)</th>
<th>LEN_SMALLINT</th>
<th>CHAR(FUNCTION)</th>
<th>LEN_INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>6</td>
<td>33</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>0</td>
<td>11</td>
</tr>
</tbody>
</table>

- Data type of value-expression is a fixed point, packed or zoned decimal: (UNSIGNED) DECIMAL, (UNSIGNED) NUMERIC.
- CHAR returns a fixed-length character string representation of the value of value-expression. Specifying a second parameter is not allowed. If value-expression has a precision of \( p \) and a scale of \( s \), the result contains \( p + 2 \) characters as follows: a blank or minus sign, depending on the sign of value-expression, \( p - s \) digits followed by a period and finally \( s \) digits. The result is left-justified.
Example:

```
SELECT VAC_TIME, CHAR(-VAC_TIME) , LENGTH(CHAR(VAC_TIME))
FROM DEMOEMPL.EMP_VACATION WHERE VAC_TIME > 300
```

<table>
<thead>
<tr>
<th>VAC_TIME</th>
<th>CHAR(FUNCTION)</th>
<th>(CONST)</th>
</tr>
</thead>
<tbody>
<tr>
<td>340.00</td>
<td>-340.0</td>
<td>33</td>
</tr>
<tr>
<td>396.00</td>
<td>-396.0</td>
<td>33</td>
</tr>
<tr>
<td>484.00</td>
<td>-484.0</td>
<td>33</td>
</tr>
</tbody>
</table>

Data type of value-expression is a floating-point data type: REAL, FLOAT or DOUBLE PRECISION

CHAR returns a fixed-length character string representation of the floating point value of value-expression. Specifying a second parameter is not allowed. The result is left justified and contains 24 characters.

Example:

```
SELECT AVGROWLENGTH, CHAR(AVGROWLENGTH) , LENGTH(CHAR(AVGROWLENGTH)) AS L24
FROM SYSTEM.TABLE WHERE NAME = 'TABLE';
```

<table>
<thead>
<tr>
<th>AVGROWLENGTH</th>
<th>CHAR(FUNCTION)</th>
<th>L24</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5600000E+02</td>
<td>2.56E2</td>
<td>24</td>
</tr>
<tr>
<td>0.0000000E+00</td>
<td>0.0E0</td>
<td>24</td>
</tr>
</tbody>
</table>

Data type of value-expression is a character data type CHAR, VARCHAR.

CHAR returns a fixed-length character string representation of the value of value-expression. An exact-numeric-literal can be specified as a second parameter, in which case it defines the length of the result. The value of exact-numeric-literal must be in the range 0-255. If the length of value-expression is lower than exact-numeric-literal the result will be padded with blanks on the right, else if the length is larger, truncation will occur and, if nonblank characters are truncated, a warning message is issued.
Example:

SELECT CHAR(NAME, 4), LENGTH(CHAR(NAME, 4)) AS LEN
FROM SYSTEM.TABLE WHERE NAME = 'TABLE';

<table>
<thead>
<tr>
<th>CHAR(FUNCTION)</th>
<th>LEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABL</td>
<td>4</td>
</tr>
</tbody>
</table>

Data type of value-expression is DATE, TIME, or TIMESTAMP.

If no format (ISO, USA, EUR, JIS) is specified for the character string, the result is returned in ISO format or, if the SQL statement is embedded in a program, the format specified in a precompiler option.

Note: For information about specifying precompiler options, see the CA IDMS SQL Programming Guide.

Parameters

ISO

Specifies that the format of the result should comply with the standard of the International Standards Organization (ISO). Formats used when ISO is specified are:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>yyyy-mm-dd</td>
<td>1990-12-15</td>
</tr>
<tr>
<td>TIME</td>
<td>hh.mm.ss</td>
<td>16.43.17</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>yyyy-mm-dd-hh.mm.ss.nnnnn</td>
<td>1990-12-15-16.43.17.1234.56</td>
</tr>
</tbody>
</table>

USA

Specifies that the format of the result should comply with the standard of the IBM USA standard. Formats used when USA is specified are:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>mm/dd/yyyy</td>
<td>12/15/1990</td>
</tr>
<tr>
<td>TIME</td>
<td>hh:mm AM</td>
<td>4:43 PM</td>
</tr>
<tr>
<td></td>
<td>hh:mm PM</td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>yyyy-mm-dd-hh.mm.ss.nnnnn</td>
<td>1990-12-15-16.43.17.1234.56</td>
</tr>
</tbody>
</table>
EUR

Specifies that the format of the result should comply with the standard of the IBM European standard. Formats used when EUR is specified are:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>dd.mm.yyyy</td>
<td>15.12.1990</td>
</tr>
<tr>
<td>TIME</td>
<td>hh.mm.ss</td>
<td>16.43.17</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>yyyy-mm-dd-hh.mm.ss.nnnnnn</td>
<td>1990-12-15-16.43.17.1234 56</td>
</tr>
</tbody>
</table>

JIS

Specifies that the format of the result should comply with the standard of the Japanese Industrial Standard Christian Era. Formats used when JIS is specified are:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>yyyy-mm-dd</td>
<td>1990-12-15</td>
</tr>
<tr>
<td>TIME</td>
<td>hh:mm:ss</td>
<td>16:43:17</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>yyyy-mm-dd-hh.mm.ss.nnnnnn</td>
<td>1990-12-15-16.43.17.1234 56</td>
</tr>
</tbody>
</table>

CHAR_LENGTH or CHARACTER_LENGTH-functions

Syntax

```
CHAR_LENGTH ( or CHARACTER_LENGTH ) ( value-expression )
```

CHAR_LENGTH (or CHARACTER_LENGTH) obtains the length of the value in `value-expression`.

The result of the CHAR_LENGTH (or CHARACTER_LENGTH) function is an integer.

The length of a value depends on its data type:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY</td>
<td>The number of bytes which contain the value</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>The actual number of characters in the string,</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>including blanks</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>The number of DBCS characters</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td></td>
</tr>
</tbody>
</table>
**COALESCE-function**

The COALESCE scalar function is identical to the VALUE scalar function, so they are listed together. See VALUE or COALESCE-function for more information.

**CONCAT-function**

**Syntax**

```
CONCAT (value-expression1, value-expression2)
```

**Parameters**

**value-expression1**

Specifies a character string value-expression. See Expansion of Value-expression for more information.

**value-expression2**

Specifies a character string value-expression. See Expansion of Value-expression for more information.

CONCAT (value-expression1, value-expression2) is equivalent to value-expression1 || value-expression2. ‘||’ is the concatenation operator and concatenates the value-expression2 to value-expression1.

value-expression1 and value-expression2 can be of BINARY, CHARACTER, VARCHAR, GRAPHICS, or VARGRAPHIC data type.

If any of the value-expressions are null, the result is the null value.

**Example**

The following statement returns 'A1B2C3':

```
SELECT CONCAT(CONCAT('A1', 'B2'), 'C3')
               FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**CONVERT-function**

**Syntax**

```
CONVERT ( value-expression, data-type )
```
CONVERT is semantically equivalent with CAST. See CAST-function for more information.

**Example**

The following statement returns 1.1999999999999999E+00:

```
SELECT CONVERT (1.2, DOUBLE PRECISION)
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**COS-function**

**Syntax**

```
COS ( value-expression )
```

COS returns the cosine of the value-expression, which must be an angle expressed in radians. COS is the inverse function of the ACOS function.

The value-expression must be of any numeric data type. It is converted to a double precision floating-point number for processing by this function.

The result of the function is a double precision floating-point number. If the value-expression is null, the result is the null value. If a data error occurs, an exception is raised.

**Example**

The following statement returns 7.0000000000000037E-01:

```
SELECT COS(7.9539883018414370E-01 )
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**COSH-function**

**Syntax**

```
COSH ( value-expression )
```

COSH returns the hyperbolic cosine of the value-expression, which must be an angle expressed in radians.

The value-expression must be of any numeric data type. It is converted to a double precision floating-point number for processing by this function.

The result of the function is a double precision floating-point number. If the value-expression is null, the result is the null value. If a data error occurs, an exception is raised.

Example

The following statement returns 1.5430806348152437E+00:

```sql
SELECT COSH (1)
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

COT-function

Syntax

```
COT (value-expression)
```

COT returns the cotangent of the value-expression, which must be an angle expressed in radians.

The value-expression must be of any numeric data type. It is converted to a double precision floating-point number for processing by this function.

The result of the function is a double precision floating-point number. If the value-expression is null, the result is the null value. If a data error occurs, an exception is raised.

Example

The following statement returns 1.0000000000000000E+00:

```sql
SELECT COT(PI() / 4)
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

CURDATE-function

Syntax

```
CURDATE()
```

CURDATE returns the current date.
CURDATE is equivalent to the special-register CURRENT DATE. See Expansion of Special-register for more information.

Example

The following statement returns the current date two times:

```
SELECT CURDATE(), CURRENT DATE
    FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**CURTIME-function**

Syntax

```sql
CURTIME ()
```

CURTIME is equivalent to the special-register CURRENT TIME. See Expansion of Special-register for more information.

Example

The following statement returns the current time two times:

```
SELECT CURTIME(), CURRENT TIME
    FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**DATABASE-function**

Syntax

```sql
DATABASE ()
```

DATABASE is equivalent to the special-register CURRENT DATABASE. See Expansion of Special-register for more information.

Example

The following statement returns the current database 'SYSDICT':

```
SELECT DATABASE()
    FROM SYSTEM.SCHEMA WHERE NAME='SYSTEM';
```

**DATE-function**

Syntax

```sql
DATE ( value-expression )
```
DATE obtains the date from the value in value-expression.

The result of the DATE function depends on the type of value in value-expression:

<table>
<thead>
<tr>
<th>Value-expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP value</td>
<td>The date part of the timestamp</td>
</tr>
<tr>
<td>DATE value</td>
<td>The date</td>
</tr>
<tr>
<td>Numeric value</td>
<td>The date that is n-1 days after January 1, 0001, where n is the number that would result if the INTEGER function were applied to value-expression</td>
</tr>
<tr>
<td>Character string in the form yyyynnn where yyyy denotes a year and nnn is in the range 001 to 366 denoting a day of that year</td>
<td>The date represented by the character string</td>
</tr>
</tbody>
</table>

**DAY or DAYOFMONTH-function**

**Syntax**

```sql
DAY (value-expression) DAYOFMONTH
```

DAY obtains the day part of the value in value-expression.

**Value-expression** must be a date, timestamp, or date duration.

The result of the DAY function is an integer, as shown in the next table.

<table>
<thead>
<tr>
<th>Value-expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP value</td>
<td>1 to 31 (the day part of the timestamp)</td>
</tr>
<tr>
<td>DATE value</td>
<td>1 to 31 (the day part of the date)</td>
</tr>
<tr>
<td>Date duration</td>
<td>The day part of the value (an integer in the range -99 to 99 with the same sign as value-expression if the result is not 0)</td>
</tr>
</tbody>
</table>

**Example**

The following statement returns 25:

```sql
SELECT DAYOFMONTH ('2002-12-25')
    FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```
**DAYNAME-function**

**Syntax**

```sql
DAYNAME ( value-expression )
```

DAYNAME returns a character string containing the English name of the day specified by value-expression.

*value-expression* must be a DATE or TIMESTAMP data type or must be a CHARACTER or VARCHAR data type and represent a valid string representation of a date or timestamp.

The result is of CHARACTER(12) data type.

The result is null if value-expression is null.

**Example**

The following statement returns the names of all days from now to now + 6 days
"Tuesday Wednesday Thursday Friday Saturday Sunday Monday:"

```sql
SELECT  DAYNAME(NOW() + 0 DAY),
       DAYNAME(NOW() + 1 DAY),
       DAYNAME(NOW() + 2 DAY),
       DAYNAME(NOW() + 3 DAY),
       DAYNAME(NOW() + 4 DAY),
       DAYNAME(NOW() + 5 DAY),
       DAYNAME(NOW() + 6 DAY)
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**DAYOFWEEK-function**

**Syntax**

```sql
DAYOFWEEK ( value-expression )
```

DAYOFWEEK returns the day of the week where 1 is Sunday and 7 is Saturday.

*value-expression* must be a DATE or TIMESTAMP data type or must be a CHARACTER or VARCHAR data type and represent a valid string representation of a date or timestamp.

The result is an INTEGER data type.

The result is null if *value-expression* is null.

**Example**

The following statement returns 4, which represents Wednesday:

```sql
SELECT DAYOFWEEK ('2002-12-25')
       FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**DAYOFYEAR-function**

**Syntax**

```
SIGN DAYOFYEAR ( value-expression )
```

DAYOFYEAR returns the day of the year where 1 is January 1.

*value-expression* must be a DATE or TIMESTAMP data type or must be a CHARACTER or VARCHAR data type and represent a valid string representation of a date or timestamp.

The result is an INTEGER data type and in the range of 1 to 366.

The result is null if *value-expression* is null.

**Example**

The following statement returns 365:

```sql
SELECT DAYOFYEAR ('2002-12-31')
       FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**DAYS-function**

**Syntax**

```
SIGN DAYS ( value-expression )
```

DAYS returns the number of days between two dates.

*value-expression* must be a DATE or TIMESTAMP data type or must be a CHARACTER or VARCHAR data type and represent a valid string representation of a date or timestamp.

The result is an INTEGER data type.

The result is null if *value-expression* is null.
DAYS obtains an integer representation of the date in `value-expression`.

`Value-expression` must be a date, timestamp, or valid character string representation of a date.

The result of the DAYS function is \( d + 1 \) days from January 1, 0001, where \( d \) is the date that would result if the DATE function were applied to `value-expression`.

### DECIMAL-function

**Syntax**

```
DECIMAL ( value-expression integer precision , scale )
```

DECIMAL obtains a decimal representation of the value in `value-expression`.

`Value-expression` must be numeric.

The result of the DECIMAL function is a decimal number. The following table shows the default precision and scale of the result if `precision` is not specified.

<table>
<thead>
<tr>
<th>Value-expression data type</th>
<th>Default precision</th>
<th>Default scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>LONGINT</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>INTEGER</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Other numeric data types</td>
<td>Same as value-expression</td>
<td>Same as value-expression</td>
</tr>
</tbody>
</table>

**Parameters**

- `precision`
  
  Specifies the number of digits in the result. `Precision` must be an integer in the range of 1 to 31.

- `scale`
  
  Specifies the number of digits to the right of the decimal point in the result. `Scale` must be an integer in the range of 0 to the value of `precision`.

### DEGREES-function

**Syntax**

```
DEGREES ( value-expression )
```

DEGREES returns the number of degrees calculated from the value-expression expressed in radians. The value-expression must be of any numeric data type. It is converted to a double precision floating-point number for processing by this function.

The result of the function is a double precision floating-point number.

If the value-expression is null, the result is the null value. If a data error occurs, an exception is raised.

Example

The following statement returns 8.9999999999999985E+01:

```
SELECT DEGREES(PI() / 2)
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**DIGITS-function**

**Syntax**

```
DIGITS ( value-expression )
```

DIGITS obtains a character string representation of the value in value-expression.

Value-expression must be an integer or decimal number.

The result of the DIGITS function is a fixed-length string of digits that represents the absolute value of value-expression and ignores scale. Thus, the result has no sign and no decimal point. The result includes leading zeros.

The length of the result is:

- 5 if value-expression has a data type of SMALLINT
- 10 if value-expression has a data type of INTEGER
- 19 if value-expression has a data type of LONGINT
- The precision of value-expression if it contains a decimal number

**EXP-function**

**Syntax**

```
EXP ( value-expression )
```
EXP returns a value that is calculated as the base of the natural logarithm (e), raised to a power specified by the value-expression. EXP is the inverse function of LOG.

The value-expression must be of any numeric data type. It is converted to a double precision floating-point number for processing by this function.

The result of the function is a double precision floating-point number. If the value-expression is null, the result is the null value. If a data error occurs, an exception is raised.

Example

The following statement returns 2.7182818284590451E+00:

```
SELECT   EXP (1)
  FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**FLOAT-function**

**Syntax**

```
FLOAT ( value-expression )
```

FLOAT obtains a floating-point representation of the value in value-expression.

Value-expression must be a number.

The result of the FLOAT function is a double precision floating-point number. It is the same number that would result if value-expression were assigned to a column with a data type of DOUBLE PRECISION.

**FLOOR-function**

**Syntax**

```
FLOOR ( value-expression )
```
FLOOR returns the largest integer value that is less than or equal to the value-expression.

The value-expression must be of any numeric data type.

The result of the function has the same data type as the value-expression except that the scale is 0 if the value-expression is of type (UNSIGNED) DECIMAL or (UNSIGNED) NUMERIC. For example, a value-expression with a data type of NUMERIC (3,2) results in NUMERIC(3,0). If the value-expression is null, the result is the null value. If a data error occurs, an exception is raised.

Example

The following statement returns 12 1.0000000000000000E+00 -13

```
SELECT FLOOR(12.55), FLOOR(123.1E-2), FLOOR(-12.55)
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

HEX-function

Syntax

```sql
HEX (value-expression)
```

HEX obtains a hexadecimal representation of the value in value-expression.

The result of the HEX function is a character string of hexadecimal digits. The resulting value and length of the result field depend on the data-type of value-expression. For a character operand, the first two digits represent the first byte of value-expression, the next two digits represent the second byte of value-expression, and so on.

The length of the result is limited to 32,767 digits.

If value-expression is a numeric character string, its length will vary as noted below. If it is neither a numeric nor a graphic character string, the length of the result is twice the length of value-expression (as defined by the length function) or twice the maximum length of a varying-length string. If value-expression is a graphic character string, the length of the result is four times the length of value-expression (four times the maximum length of a varying-length string)

If value-expression is a fixed-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 with a maximum length equal to:

- Twice the fixed or maximum length.
- Four times the fixed or maximum length
If value-expression is some kind of numeric data-type, the length of the resulting character string depends on the data-type in the following way:

- **BINARY**—twice the length of value-expression.
- **(UNSIGNED) DECIMAL**—twice the length of the (UNSIGNED) DECIMAL value (The length of an (UNSIGNED) DECIMAL value is equal to precision plus 1, divided by 2).
- **DOUBLE PRECISION**—twice the length of the DOUBLE PRECISION value (16 bytes, for example).
- **FLOAT**—twice the length of the FLOAT value (If precision is less than or equal to 24, the length of a FLOAT value is 4 bytes. If precision is greater than 24, the length of a FLOAT value is 8 bytes).
- **SMALLINT**—twice the length of a SMALLINT value (4 bytes, for example).
- **INTEGER**—twice the length of an INTEGER value (8 bytes, for example).
- **LONGINT/BIGINT**—twice the length of a LONGINT/BIGINT value (16 bytes, for example).
- **(UNSIGNED) NUMERIC**—twice the length of the (UNSIGNED) NUMERIC value (The length of an (UNSIGNED) NUMERIC value is equal to the precision).
- **REAL**—twice the length of the REAL value (8 bytes, for example).

If value-expression is a numeric constant, the following rules apply:

- If the value-expression is less than or equal to the maximum positive number that can be stored in a fullword (2147483647), the result will always be an 8 character hex string
  
  **Example:** HEX(2147483647) will display as 7FFFFFFF

- All other numeric constants will be displayed as packed decimal, like for the DECIMAL data-type
  
  **Example:** HEX(2147483648) will display as 02147483648C

**Note:** Since HEX(00000000) will return 00000000, it will never match a value of HEX(decimal-field), even if the value of that decimal field is 0. In this case, one should use the CAST on the numeric constant so that the value types are consistent

**Example:** Suppose the decimal-field is defined as DECIMAL(8) and contains 0, the following WHERE clause will include the row in the result table:

```
SELECT decimal-field, HEX(decimal-field) FROM rec-name
WHERE HEX(decimal-field) = HEX(CAST(0 AS DECIMAL(8)));
```

### HOUR-function

**Syntax**

```
HOUR ( value-expression )
```
HOUR obtains the hours part of the value in **value-expression**.

**Value-expression** must be a time, timestamp, or time duration.

The result of the HOUR function is an integer, as shown in the following table.

<table>
<thead>
<tr>
<th>Value-expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP value</td>
<td>0 to 24 (the hours part of the timestamp)</td>
</tr>
<tr>
<td>TIME value</td>
<td>0 to 24 (the hours part of the time)</td>
</tr>
<tr>
<td>Time duration</td>
<td>The time part of the value (an integer in the range -99 to 99 with the same sign as <strong>value-expression</strong> if the result is not 0)</td>
</tr>
</tbody>
</table>

**IFNULL-function**

**Syntax**

```
IFNULL ( value-expression1, value-expression2 )
```

**Parameters**

*value-expression1*

Specifies a value-expression. See Expansion of Value-expression.

*value-expression2*

Specifies a value-expression. See Expansion of Value-expression.

IFNULL returns the first value-expression that is not null. IFNULL is similar to the VALUE and COALESCE scalar functions with the exception that IFNULL is limited to only two value-expressions instead of multiple value-expressions.

**Note:** For more information, see VALUE or COALESCE-function.

**Example**

The following statement will show '**NULL**' for any row with a null value for SEGMENT, otherwise the name of the segment will be shown:

```
SELECT SCHEMA, NAME, IFNULL (SEGMENT, '**NULL**')
FROM SYSTEM.TABLE
```

**INSERT-function**

**Syntax**

```
INSERT ( value-expression1, start, length, value-expression2 )
```
Parameters

value-expression1

Specifies a character string value-expression. See Expansion of Value-expression.

value-expression2

Specifies a character string value-expression. See Expansion of Value-expression.

start

Specifies a numeric value-expression. See Expansion of Value-expression.

length

Specifies a numeric value-expression. See Expansion of Value-expression.

INSERT returns a string constructed from value-expression1, where beginning at start, length characters have been deleted and value-expression2 has been inserted.

value-expression1 specifies the source string and must be a CHARACTER or VARCHAR data type. If the length of value-expression1 is 0, the result the null value.

Start must be of any numeric data type, but only the integer part is considered. The integer part of start specifies the starting point within value-expression1 where the deletion of characters and the insertion of value-expression2 is to begin. The integer part of start must be in the range of 1 to the length of value-expression1 plus one.

Length must be of any numeric data type, but only the integer part is considered. The integer part of length specifies the number of characters that are to be deleted from value-expression1, starting at start. The integer part of length must be in the range of 0 to the length of value-expression1.

value-expression2 specifies the string to be inserted into value-expression1, starting at start. The string to be inserted must be a CHARACTER or VARCHAR data type.

The result is always of VARCHAR data type.

The length of the result is given by the following formula:

\[
\text{LENGTH(value-expression1)} + \text{LENGTH(value-expression2)} - \min(\text{length}, \text{LENGTH(value-expression1)} - \text{start} + 1)
\]

If both start and length are constants, the maximum length of the result is calculated during compilation of the INSERT invocation using the above formula, otherwise the maximum length is 8000.

The result is null if either value-expression1 or value-expression2 is null. If the insert cannot be done, because of invalid parameters, an exception is raised.
Example 1

The following statement appends the string 'DEF' to the string 'ABC' giving 'ABCDEF':

```
SELECT INSERT ('ABC', 4, 0, 'DEF')
    FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

Because both the start and length parameters of the INSERT function are constants, the maximum length of the result VARCHAR string is 6.

Example 2

The following statement prefixes the string 'DEF' with the string 'ABC' giving 'ABCDEF':

```
SELECT SUBSTR(ININSERT ('DEF', 1*1, 0,'ABC'), 1, 20)
    FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

Because the start position is not a constant, but an expression, the maximum length of the result VARCHAR string of INSERT is 8000. The SUBSTR function is used to limit the final result to 20 characters.

Example 3

The following statement replaces the character at position 3 in string 'ABCDEF' with the string 'XYZ' returning 'ABXYZDEF':

```
SELECT INSERT ('ABCDEF', 3, 1, 'XYZ')
    FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

Because both the start and length parameters of the INSERT function are constants, the maximum length of the result VARCHAR string is 8.

**INTEGER-function**

**Syntax**

```
INTEGER ( value-expression )
```

INTEGER obtains an integer representation of the value in *value-expression*.

*Value-expression* must be a number. The whole number part of *value-expression* must be in the range of integers.

The result of the INTEGER function is the same number that would result if *value-expression* were assigned to a column with a data type of INTEGER.
LEFT-function

Syntax

```
LEFT ( value-expression, length )
```

Parameters

value-expression

Specifies a character string value-expression. See Expansion of Value-expression.

length

Specifies a numeric value-expression. See Expansion of Value-expression.

LEFT obtains a substring of the value in value-expression, starting with character position 1.

Value-expression must be a character or graphics string.

The result of the LEFT function is a character string when value-expression is a character string; the result is a graphics string when value-expression is a graphics string.

Length is a value expression that must be an integer not less than 1, and must not exceed the length of the string in value-expression. (The length of a value with a data type of VARCHAR or VARGRAPHIC is its maximum length.)

If the substring is less than the specified length, CA IDMS pads the result with blanks.

If length is not specified, the substring begins at start and ends at the end of the string.

If length is null, the result of the function is null.

LEFT ( value-expression, length ) is equivalent to SUBSTR ( value-expression, 1, length ).

LENGTH-function

Syntax

```
LENGTH ( value-expression )
```

LENGTH obtains the length of the value in value-expression.

The result of the LENGTH function is an integer.

The length of a value depends on its data type, as shown in the following table.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE PRECISION</td>
<td>8 bytes</td>
</tr>
</tbody>
</table>
### Data type | Length
---|---
FLOAT | 4 bytes if precision <= 24
 | 8 bytes if precision > 24
REAL | 4 bytes
BINARY | The number of bytes containing the value
CHARACTER | The actual number of characters in the string, including blanks
VARCHAR | The actual number of characters in the string, including blanks
DATE | 10 bytes
TIME | 8 bytes
TIMESTAMP | 26 bytes
DECIMAL | The number of bytes containing the value
INTEGER | 4 bytes
LONGINT | 8 bytes
NUMERIC | The number of bytes containing the value
SMALLINT | 2 bytes
UNSIGNED DECIMAL | The number of bytes containing the value
UNSIGNED NUMERIC | The number of bytes containing the value
GRAPHIC | The number of DBCS characters
VARGRAPHIC | The number of DBCS characters

#### LOCATE-function

**Syntax**

```sql
LOCATE (value-expression1, value-expression2 [start])
```
LOCATE returns an integer value representing the location of the first value expression within the second value expression. The value expressions must each be either CHARACTER or VARCHAR. If the first value expression does not appear in the second value expression, the result of LOCATE is 0. Otherwise, the result of LOCATE is the byte position of the first matching character within the second string.

Start specifies the character position within the second value-expression at which the search for the first value-expression is to start.

Start is a value-expression that must be an integer not greater than the length of the string in the value-expression2.

**LOG-function**

**Syntax**

```sql
LOG (value-expression)
```

LOG returns a value that is calculated as the natural logarithm of value-expression. LOG is the inverse function of EXP.

The value-expression must be of any numeric data type. It is converted to a double precision floating-point number for processing by this function.

The result of the function is a double precision floating-point number. If the value-expression is null, the result is the null value. If a data error occurs, an exception is raised.

**Example**

The following statement returns `1.0986122886681095E+00`:

```sql
SELECT LOG (3)
FROM SYSTEM.SCHEMAM WHERE NAME = 'SYSTEM'
```
LOG10-function

Syntax

►► LOG10 ( value-expression ) ←←

LOG10 returns a value that is calculated as the base 10 logarithm of value-expression.

The value-expression must be of any numeric data type. It is converted to a double precision floating-point number for processing by this function.

The result of the function is a double precision floating-point number. If the value-expression is null, the result is the null value. If a data error occurs, an exception is issued.

Example

The following statement returns 3.0000000000000000E+00:

SELECT LOG (1000)
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';

LOWER or LCASE-function

Syntax

►► LOWER ( value-expression ) ←←

LOWER operates on CHARACTER or VARCHAR value-expressions. The result is a string of equal length where all upper case characters have been folded into lower case.

Example

The following statement returns 'joe carpenter':

SELECT LCASE('JOE CARPENTER')
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';

LTRIM-function

Syntax

►► LTRIM ( value-expression ) ←←

LTRIM removes leading blanks from value-expression.

Value-expression must be a character string. It is the equivalent of TRIM (leading FROM value-expression).
MICROSECOND-function

Syntax

```
MICROSECOND ( value-expression )
```

MICROSECOND obtains the microsecond part of the value in `value-expression`.

`Value-expression` must be a timestamp.

The result of the MICROSECOND function is an integer in the range 0 to 999999, representing the microsecond part of the timestamp.

MINUTE-function

Syntax

```
MINUTE ( value-expression )
```

MINUTE obtains the minutes part of the value in `value-expression`.

`Value-expression` must be a time, timestamp, or time duration.

The result of the MINUTE function is an integer, as shown in the following table.

<table>
<thead>
<tr>
<th>Value-expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP value</td>
<td>0 to 59 (the minutes part of the timestamp)</td>
</tr>
<tr>
<td>TIME value</td>
<td>0 to 59 (the minutes part of the time)</td>
</tr>
<tr>
<td>Time duration</td>
<td>The minute part of the value (an integer in the range -99 to 99 with the same sign as <code>value-expression</code> if the result is not 0)</td>
</tr>
</tbody>
</table>

MOD-function

Syntax

```
MOD ( value-expression1, value-expression2 )
```
Parameters

`value-expression1`

Specifies a numeric value-expression. See Expansion of Value-expression.

`value-expression2`

Specifies a numeric value-expression. See Expansion of Value-expression.

MOD returns the remainder of dividing `value-expression1` by `value-expression2` using the formula:

\[ \text{MOD}(v1, v2) = v1 - \text{Truncated Integer}(v1/v2) \times v2 \]

with `Truncated Integer(v1 / v2)` the truncated integer result of the division.

Both value-expressions must be of any numeric data type. The second value-expression cannot be zero.

If the value-expression is null, the result is the null value. If a data error occurs, an exception is issued.

The data type of the result follows these rules:

- If both value-expressions are INTEGER or SMALLINT, the data type of the result is INTEGER.
- If one of the value-expressions is LONGINT, and the other is INTEGER, SMALLINT, or LONGINT, the data type of the result is also LONGINT.
- If one value-expression is an INTEGER, SMALLINT, or LONGINT and the other is an (UNSIGNED) DECIMAL or (UNSIGNED) NUMERIC, the data type of the result is (UNSIGNED) DECIMAL or (UNSIGNED) NUMERIC with the same precision and scale as the (UNSIGNED) DECIMAL or (UNSIGNED) NUMERIC value-expression.
- If both value-expressions are (UNSIGNED) DECIMAL or (UNSIGNED) NUMERIC, the data type of the result is equal to the data type of `value-expression1`. The precision and scale of the result are given by the following formulas:

  \[
  \text{Prec. result} = \min(\text{prec.1-scale.1}, \text{prec.2-scale.2}) + \max(\text{scale.1}, \text{scale.2})
  \]
  \[
  \text{Scale.result} = \max(\text{scale1, scale2})
  \]

- If either value-expression is a floating-point number, REAL, FLOAT, or DOUBLE PRECISION, the data type of the result is double precision floating-point.

The processing of this function is always done in floating-point. Both value-expressions are converted to double precision floating-point numbers.
Example 1

The following statement returns 1:

```
SELECT MOD(10, 3)
    FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

Example 2

The following statement returns 1.0000000000000000E+00:

```
SELECT MOD(1E0, 3)
    FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

Example 3

The following statement returns 1.0:

```
SELECT MOD(10.0, 3)
    FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

Example 4

The following statement returns 1.00:

```
SELECT MOD(10.00, 3)
    FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

MONTH-function

Syntax

```
► MONTH ( value-expression ) ◄
```

MONTH obtains the month part of the value in `value-expression`.

`Value-expression` must be a date, timestamp, or date duration.

The result of the MONTH function is an integer, as shown in the following table.

<table>
<thead>
<tr>
<th>Value-expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP value</td>
<td>1 to 12 (the month part of the timestamp)</td>
</tr>
<tr>
<td>DATE value</td>
<td>1 to 12 (the month part of the date)</td>
</tr>
<tr>
<td>Date duration</td>
<td>The date part of the value (an integer in the range -99 to 99 with the same sign as <code>value-expression</code> if the result is not 0)</td>
</tr>
</tbody>
</table>
MONTHNAME-function

Syntax

```
MONTHNAME ( value-expression )
```

MONTHNAME returns a character string containing the English name of the month specified by value-expression.

value-expression must be a DATE or TIMESTAMP data type or must be a CHARACTER or VARCHAR data type and represent a valid string representation of a date or timestamp.

The result is of CHARACTER(12) data type.

The result is null if value-expression is null.

Example

The following statement returns the names of all months from now to now + 11 months "January February March April May June July August September October November December:"

```
SELECT   MONTHNAME(NOW() + 0  MONTH),
         MONTHNAME(NOW() + 1  MONTH),
         MONTHNAME(NOW() + 2  MONTH),
         MONTHNAME(NOW() + 3  MONTH),
         MONTHNAME(NOW() + 4  MONTH),
         MONTHNAME(NOW() + 5  MONTH),
         MONTHNAME(NOW() + 6  MONTH),
         MONTHNAME(NOW() + 7  MONTH),
         MONTHNAME(NOW() + 8  MONTH),
         MONTHNAME(NOW() + 9  MONTH),
         MONTHNAME(NOW() + 10  MONTH),
         MONTHNAME(NOW() + 11  MONTH),
         MONTHNAME(NOW() + 12  MONTH)
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM'
```

NOW-function

Syntax

```
NOW ( )
```

Chapter 5: Functions  155
NOW is equivalent to the special-register CURRENT TIMESTAMP. See Expansion of Special-register for more information.

Example

The following statement returns the current date and time two times:

SELECT NOW(), CURRENT TIMESTAMP
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';

**OCTET_LENGTH-function**

**Syntax**

```
OCTET_LENGTH ( value-expression )
```

OCTET_LENGTH obtains the length in bytes of the value in **value-expression**.

The result of the OCTET_LENGTH function is an integer.

The length of a value depends on its data type:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE PRECISION</td>
<td>8</td>
</tr>
<tr>
<td>FLOAT</td>
<td>4 if precision &lt;= 24</td>
</tr>
<tr>
<td></td>
<td>8 if precision &gt; 24</td>
</tr>
<tr>
<td>REAL</td>
<td>4</td>
</tr>
<tr>
<td>BINARY</td>
<td>The number of bytes which contain the value</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>The actual number of bytes in the string, including blanks</td>
</tr>
<tr>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>DATE</td>
<td>10</td>
</tr>
<tr>
<td>TIME</td>
<td>8</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>26</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>The number of bytes which contain the value</td>
</tr>
<tr>
<td>INTEGER</td>
<td>4</td>
</tr>
<tr>
<td>LONGINT</td>
<td>8</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>The number of bytes which contain the value</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>2</td>
</tr>
<tr>
<td>UNSIGNED DECIMAL</td>
<td>The number of bytes which contain the value</td>
</tr>
</tbody>
</table>
## PI-function

**Syntax**

```
PI ( )
```

PI returns the constant value of pi as a floating point value. The value returned is 3.141592653589793238.

**Example**

The following statement returns 3.1415926535897933E+00:

```
SELECT PI()
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

## POSITION-function

**Syntax**

```
POSITION ( value-expression IN value-expression )
```

POSITION returns an integer value representing the location of the first value expression within the second value expression. The value expressions must each be either CHARACTER or VARCHAR. If the first value expression does not appear in the second value expression, the result of POSITION is 0. Otherwise, the result of POSITION is the byte position of the first matching character within the second string.

It is the equivalent of LOCATE (value-expression, value-expression, 1).

## POWER-function

**Syntax**

```
POWER ( value-expression1, value-expression2 )
```

Data type | Length
---|---
UNSIGNED NUMERIC | The number of bytes which contain the value
GRAPHIC | Two times the number of DBCS characters
VARGRAPHIC | Two times the number of DBCS characters
Parameters

\textit{value-expression1}

Specifies a numeric value-expression. See Expansion of Value-expression.

\textit{value-expression2}

Specifies a numeric value-expression. See Expansion of Value-expression.

\begin{itemize}
  \item \textbf{POWER} returns the value of \textit{value-expression1} to the power of \textit{value-expression2}.
\end{itemize}

The data types of \textit{value-expression1} and \textit{value-expression2} must be numeric data types.

The internal processing of this function is done using double precision floating-point arithmetic.

The data type of the result of the function depends on the data types of \textit{value-expression1} and \textit{value-expression2}:

\begin{itemize}
  \item The result is of \texttt{INTEGER} type if \textit{value-expression1} and \textit{value-expression2} are \texttt{SMALLINT} or \texttt{INTEGER}. The result is \texttt{LONGINT} if one of the \textit{value-expression} is \texttt{LONGINT} and the other \texttt{LONGINT}, \texttt{INTEGER} or \texttt{SMALLINT}, otherwise, the result is \texttt{DOUBLE PRECISION}.
  \item The result is null if either \textit{value-expression1} or \textit{value-expression2} is null. If the calculation resulted in a data error, an exception is raised.
\end{itemize}

\textbf{Example 1}

The following statement returns the value 625:

\begin{verbatim}
SELECT POWER(25, 2)
  FROM SYSTEM.TABLE WHERE NAME = 'SCHEMA'
\end{verbatim}

\textbf{Example 2}

The following \texttt{SELECT} returns the value \texttt{6.2500000000000000E+02}:

\begin{verbatim}
SELECT POWER(25.0, 2)
  FROM SYSTEM.TABLE WHERE NAME = 'SCHEMA'
\end{verbatim}

\textbf{PROFILE-function}

\textbf{Syntax}

\begin{verbatim}
PROFILE ( value-expression )
\end{verbatim}
PROFILE obtains the value associated with an attribute of the current user session.

**Value-expression** must be a character string.

The result of the PROFILE function is a CHARACTER value with a length of 32. If **value-expression** does not correspond to an attribute keyword for the session, the function returns a null value.

**Note:** For more information about attributes of a user session, see the discussion of system profiles in *CA IDMS System Tasks and Operator Commands Guide*.

### QUARTER-function

**Syntax**

```
QUARTER ( value-expression )
```

QUARTER returns the quarter of the year in which the date, specified by **value-expression**, occurs.

**value-expression** must be a DATE or TIMESTAMP data type or must be a CHARACTER or VARCHAR data type and represent a valid string representation of a date or timestamp.

The result is an INTEGER data type and is in the range of 1 to 4.

The result is null if **value-expression** is null.

**Example**

The following statement returns 4 because December is in the last quarter of the year:

```
SELECT QUARTER('2002-12-31')
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM'
```

### RADIANS-function

**Syntax**

```
RADIANS ( value-expression )
```

RADIANS converts the value in **value-expression** from degrees to radians.
RADIANS returns the number of radians corresponding to the number of degrees specified by `value-expression`.

The `value-expression` must be of any numeric data type. It is converted to a double precision floating-point number for processing by this function.

The result of the function is a double precision floating-point number.

If the `value-expression` is null, the result is the null value. If a data error occurs, an exception is raised.

**Example**

The following statement returns 3.1415926535897931E+00, which is an approximate value of PI:

```sql
SELECT RADIANS(180)
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**RAND-function**

**Syntax**

```
```

RAND returns a random floating-point value between 0 and 1. `value-expression` is optional and specifies a seed value. If no seed value is specified, 1 will be used as seed value.

If specified, the `value-expression` must be of any numeric data type. It is converted to an INTEGER number for processing by this function.

The result of the function is a double precision floating-point number.

If a data error occurs, an exception is raised.

Within the context of an IDMS task, the optional seed value is only evaluated once during the very first call of the random generator with a seed value. The series of generated random numbers will be equal for equal seed values when executed under different IDMS tasks.

**Example**

The following statement returns random floating-point numbers between 0 and 1:

```sql
SELECT RAND (200), RAND()
FROM SYSTEM.SCHEMA;-- WHERE NAME = 'SYSTEM';
```
REPEAT-function

Syntax

REPEAT ( value-expression, count )

REPEAT returns a string constructed as count times value-expression repeated.

Parameters

value-expression

Specifies the string to be repeated and must be a CHARACTER or CHAR data type.

count

Specifies an expression of any numeric data type, but only the integer part is considered. The integer part of count specifies the number of times to repeat value-expression.

The result of the function is VARCHAR.

The length of the result is the length of expression times count. If the actual length of the result string exceeds the maximum for the return type, an error occurs.

If count is a constant, the maximum length of the result is calculated during compilation of the REPEAT function invocation, otherwise the maximum is 16000.

The result is null if either value-expression or count is null. If the repeat cannot be done because of invalid parameters, an exception is raised.

Example 1

The following statement returns 'ABCDABCDABCDABCD':

SELECT REPEAT('ABCD', 4)
    FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';

Example 2

The following statement returns a string with length 0:

SELECT REPEAT('ABCD', 0)
    FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';

Example 3

The following statement returns <null> because count is negative:

SELECT REPEAT('ABCD', -2)
    FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
REPLACE-function

Syntax

\[
\text{REPLACE} \left( \text{value-expression1}, \text{value-expression2}, \text{value-expression3} \right)
\]

Parameters

\text{value-expression1}

Specifies a character string value-expression. See Expansion of Value-expression.

\text{value-expression2}

Specifies a character string value-expression. See Expansion of Value-expression.

\text{value-expression3}

Specifies a character string value expression. See Expansion of Value-expression.

REPLACE replaces all occurrences of \text{value-expression2} in \text{value-expression1} with \text{value-expression3}. If \text{value-expression2} was not found in \text{value-expression1}, \text{value-expression1} is returned unchanged.

\text{value-expression1} is a non-null expression that specifies the source string.

\text{value-expression2} is a non-null expression that specifies the string to be replaced in the source string.

\text{value-expression3} is an expression that specifies the replacement string. A null value will cause \text{value-expression1} to be returned unchanged. The arguments must all have data types that are compatible with VARCHAR, that is CHARACTER or VARCHAR. The actual length of each string must be less than or equal to 8000. The data type of the result is VARCHAR and the resulting length must be less than or equal to 8000. The length of the result is given by the following formula, where \( n \) is the number of occurrences of value-expression2 in value-expression1:

\[
\text{LENGTH(value-expression1)} + (n \times \text{LENGTH(value-expression3)} - \text{LENGTH(value-expression2)})
\]

The result is null if either \text{value-expression1}, \text{value-expression2}, or \text{value-expression3} is null. If the replace cannot be done because of invalid parameters, that is, in case one or more of the lengths exceed the limit, an exception is raised.
Example 1

In this example, the result is '$$$$123.0$$$$99'.

Replace all characters '*' in the string '***123.0***99' with '$$'.

```
SELECT REPLACE('***123.0***99', '*', '$$')
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM'
```

Example 2

List the departments of the EMPSCHM.DEPARTMENT table in alphabetical order, but ignore any spaces when sorting. The REPLACE function removes all spaces in the SORT_NAME column of the result.

```
SELECT *, REPLACE (DEPT_NAME_0410, ' ', '') SORT_NAME
FROM EMPSCHM.DEPARTMENT
ORDER BY SORT_NAME;
```

Example 3

In this example, the result is 'LOTS OF **FOO**LISH TALK'.

Replace string 'FOO' in the string 'LOTS OF FOOLISH TALK' with '***FOO***'.

```
SELECT REPLACE('LOTS OF FOOLISH TALK', 'FOO', '***FOO***')
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM'
```

**RIGHT-function**

**Syntax**

```
RIGHT ( value-expression, count )
```

RIGHT returns a string constructed from the specified number of rightmost count characters of value-expression.

**Parameters**

**value-expression**

Specifies the string from which the result is constructed and must be a CHARACTER or VARCHAR data type.

**count**

Specifies of any numeric data type, but only the integer part is considered. The integer part of count specifies the length of the result. The integer part of count must be an integer between 0 and n, where n is the length of value-expression.
The result is null if either `value-expression1` or `count` is null. If `count` is larger than the length of `value-expression1`, or if an error occurs, an exception is raised.

**Example 1**

The following statement returns the string 'CD':

```sql
SELECT RIGHT ('ABCD', 2 )
   FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**Example 2**

The following statement returns a string with length 0:

```sql
SELECT RIGHT ('ABCD', 0 )
   FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

---

### ROUND-function

**Syntax**

```
ROUND ( value-expression1, value-expression2 )
```

**Parameters**

- **value-expression1**
  
  Specifies a numeric value-expression. See Expansion of Value-expression.

- **value-expression2**
  
  Specifies a numeric value-expression. See Expansion of Value-expression.

`ROUND` returns `value-expression1` rounded to `value-expression2` places to the right of the decimal point if `value-expression2` is positive, or, to the left of the decimal point if `value-expression2` is zero or negative.

The `value-expression1` must be of any numeric data type.
The \textit{value-expression2} must be of any numeric data type but will be converted internally to INTEGER. The integer value of \textit{value-expression2} specifies the number of places to the right of the decimal point for the result if \textit{value-expression2} is not negative. If \textit{value-expression2} is negative, \textit{value-expression1} is rounded to $1 + \text{the absolute integer value of value-expression2}$ number of places to the left of the decimal point. If the absolute integer value of \textit{value-expression2} is larger than the number of digits to the left of the decimal point, the result is 0.

If \textit{value-expression1} is positive, rounding is to the next higher positive number. If \textit{value-expression1} is negative, rounding is to the next lower negative number.

The result of the function has the same data type and attributes as the \textit{value-expression1} except that the precision is increased by one if the \textit{value-expression} is of (UNSIGNED) DECIMAL or (UNSIGNED) NUMERIC data type and the precision is less than 31.

If any of the \textit{value-expression} are null, the result is the null value. If a data error occurs, an exception is raised.

**Example 1**

The following statement returns 627.46380 627.46400 627.46000 50000 627.00000 630.00000 600.00000: 1000.00000 0.00000:

\[
\text{SELECT} \quad \text{ROUND}(627.46381, 4), \\
\text{ROUND}(627.46381, 3), \\
\text{ROUND}(627.46381, 2), \\
\text{ROUND}(627.46381, 1), \\
\text{ROUND}(627.46381, 0), \\
\text{ROUND}(627.46381, -1), \\
\text{ROUND}(627.46381, -2), \\
\text{ROUND}(627.46381, -3), \\
\text{ROUND}(627.46381, -4) \\
\text{FROM} \text{SYSTEM.SCHEMA \ WHERE NAME = 'SYSTEM';}
\]

**Example 2**

The following statement returns -627.46380 -627.46400 -627.46000 -627.50000 -627.00000 -630.00000 -600.00000 : 0

\[
\text{SELECT} \quad \text{ROUND}(-627.46381, 4), \\
\text{ROUND}(-627.46381, 3), \\
\text{ROUND}(-627.46381, 2), \\
\text{ROUND}(-627.46381, 1), \\
\text{ROUND}(-627.46381, 0), \\
\text{ROUND}(-627.46381, -1), \\
\text{ROUND}(-627.46381, -2) \\
\text{FROM} \text{SYSTEM.SCHEMA \ WHERE NAME = 'SYSTEM';}
\]
RTRIM-function

Syntax

```sql
RTRIM ( value-expression )
```

RTRIM removes trailing blanks from value-expression. Value-expression must be a character string. It is the equivalent of TRIM (trailing FROM value-expression).

SECOND-function

Syntax

```sql
SECOND ( value-expression )
```

SECOND obtains the seconds part of the value in value-expression.

Value-expression must be a time, timestamp, or time duration.

The result of the SECOND function is an integer, as shown in the following table.

<table>
<thead>
<tr>
<th>Value-expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP value</td>
<td>0 to 59 (the seconds part of the timestamp)</td>
</tr>
<tr>
<td>TIME value</td>
<td>0 to 59 (the seconds part of the time)</td>
</tr>
<tr>
<td>Time duration</td>
<td>The time part of the value (an integer in the range -99 to 99 with the same sign as value-expression if the result is not 0)</td>
</tr>
</tbody>
</table>

SIGN-function

Syntax

```sql
SIGN ( value-expression )
```
SIGN returns an indicator of the sign of \textit{value-expression}. The possible values for the indicator are:

-1 if \textit{value-expression} is less than zero
0 if \textit{value-expression} is zero
1 if \textit{value-expression} is greater than zero

\textit{value-expression} must be of any numeric data type except (UNSIGNED) DECIMAL or (UNSIGNED) NUMERIC with a scale and precision of 31. The data type and attributes of the result of the function are the same as the \textit{value-expression} except when the \textit{value-expression} is (UNSIGNED) DECIMAL or (UNSIGNED) NUMERIC. The precision is incremented if the \textit{value-expression}'s precision and scale are equal. This is to allow for the return values of the function.

If the \textit{value-expression} is null, the result is the null value. If a data error occurs, an exception is raised.

**Example**

The following statement returns -1 0 1:

```
SELECT SIGN (1 - 10), SIGN (0), SIGN (1 + 10)
  FROM SYSTEM.TABLE WHERE NAME = 'SYSTEM'
```

### SIN-function

**Syntax**

\[ \text{SIN} \left( \text{value-expression} \right) \]

\text{SIN} returns the sine of the \textit{value-expression}, which must be an angle expressed in radians. \text{SIN} is the inverse function of the ASIN function.

The \textit{value-expression} must be of any numeric data type. It is converted to a double precision floating-point number for processing by this function.

The result of the function is a double precision floating-point number. If the \textit{value-expression} is null, the result is the null value. If a data error occurs, an exception is raised.

**Example**

The following statement returns 1.0000000000000000E+00:

```
SELECT SIN( PI() / 2)
  FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```
**SINH-function**

**Syntax**

```sql
SINH (value-expression)
```

SINH returns the hyperbolic sine of the `value-expression`, which must be an angle expressed in radians.

The `value-expression` must be of any numeric data type. It is converted to a double precision floating-point number for processing by this function.

The result of the function is a double precision floating-point number. If the `value-expression` is null, the result is the null value. If a data error occurs, an exception is raised.

**Example**

The following statement returns 1.1548739357257750E+01:

```sql
SELECT SIN(PI())
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**SPACE-function**

**Syntax**

```sql
SPACE (value-expression)
```

SPACE returns a character string that consists of `value-expression` number of blanks. `value-expression` is of any numeric data type, but only the integer part is considered. The integer part specifies the number of blanks that makes up the result, and it must be between 0 and 30000.

The result is of VARCHAR data type. The length of the result is the integer part of `value-expression`.

If `value-expression` is a constant, the maximum length of the result is calculated during compilation of the SPACE function invocation, otherwise the maximum is 30000.

The result is null if `value-expression` is null. An error occurs if `value-expression` is larger than 30000.

**Example**

The following statement returns 10 blanks:

```sql
SELECT SPACE (10)
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```
SQRT-function

Syntax

```
SQRT ( value-expression )
```

SQRT returns the square root of the `value-expression`. The `value-expression` must be of any numeric data type. It is converted to a double precision floating-point number for processing by this function.

The result of the function is a double precision floating-point number. If the `value-expression` is null, the result is the null value. If a data error occurs, an exception is raised.

Example 1

The following statement returns 4.0000000000000000E+00:

```
SELECT  SQRT(16)
        FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

Example 2

The following statement returns <null> because the square root of a negative number does not exist:

```
SELECT  SQRT(-16)
        FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

SUBSTR or SUBSTRING-function

Syntax

```
SUBSTR ( value-expression, start, length )
```

```
SUBSTRING ( value-expression FROM start FOR length )
```

SUBSTR or SUBSTRING obtains a substring of the value in `value-expression`.

Value-expression must be a character or graphics string.

The result of the SUBSTR function is a character string when `value-expression` is a character string; the result is a graphics string when `value-expression` is a graphics string.
Parameters

start

Specifies the position of the first character of the result. 

Start is a value expression that must be an integer less than or equal to the length of the string in value-expression.

If start is null, the result of the function is null.

length

Specifies the length of the result.

Length is a value expression that must be an integer not less than one. The sum of length and start must not exceed 1 + the length of the string in value-expression. (The length of a value with a data type of VARCHAR or VARGRAPHIC is its maximum length.)

If the substring is less than the specified length, CA IDMS pads the result with blanks.

If length is not specified, the substring begins at start and ends at the end of the string.

If length is null, the result of the function is null.

TAN-function

Syntax

TAN ( value-expression )

TAN returns the tangent of the value-expression, which must be an angle expressed in radians. TAN is the inverse function of the ATAN function.

The value-expression must be of any numeric data type. It is converted to a double precision floating-point number for processing by this function.

The result of the function is a double precision floating-point number. If the value-expression is null, the result is the null value. If a data error occurs, an exception is raised.

Example

The following statement returns 1.000000000000000E+00:

SELECT  TAN ( PI() / 4 )
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
TANH-function

Syntax

\[ \text{TANH ( value-expression )} \]

TANH returns the hyperbolic tangent of the value-expression, which must be an angle expressed in radians.

The value-expression must be of any numeric data type. It is converted to a double precision floating-point number for processing by this function.

The result of the function is a double precision floating-point number. If the value-expression is null, the result is the null value. If a data error occurs, an exception is raised.

Example

The following statement returns 6.5579420263267255E-01:

```
SELECT TANH ( PI()/4 )
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

TIME-function

Syntax

\[ \text{TIME ( value-expression )} \]

TIME obtains the time from the value in value-expression.

Value-expression must be a timestamp, time, or character string.

The result of the TIME function is a time:

<table>
<thead>
<tr>
<th>Value-expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP value</td>
<td>The time part of the timestamp</td>
</tr>
<tr>
<td>TIME value</td>
<td>That time</td>
</tr>
<tr>
<td>CHARACTER value in a valid time format</td>
<td>The time represented by the character string</td>
</tr>
</tbody>
</table>

TIMESTAMP-function

Syntax

\[ \text{TIMESTAMP ( value-expression, value-expression )} \]
TIMESTAMP obtains a timestamp from a value or pair of values.

If one value expression is specified, it must be:
- A value with the TIMESTAMP data type
- A valid character string representation of a timestamp
- An eight-character string in the form of a System/370 Store Clock value
- A 14-character string in the form `yyyyMMddhhmnnss` where `yyyy` is the year, `mm` is the month, `dd` is the day, `hh` is the hour, `nn` is the minutes, and `ss` is the seconds

Note: A timestamp represented by a 14-character string has a microsecond part of zero. The interpretation of an eight-character string is a timestamp, as discussed under Store Clock value in *IBM System/370 Principles of Operations*.

If two value expressions are specified, the first value-expression must be a date or valid character string representation of a date, and the second value-expression must be a time or valid character string representation of a time.

The result of the function is a value with the TIMESTAMP data type:
- If two value expressions are specified, the result is a timestamp with the date specified in the first value and the time specified in the second value
- If one value expression is specified and it is a character string, the result is the timestamp represented by the character string

**TRIM-function**

**Syntax**

```
TRIM ( value-expression-1 FROM value-expression-2 )
```

TRIM removes leading or trailing (or both) pad characters to be removed from a CHARACTER or VARCHAR value-expression.

The optional value expression defines the pad character to be removed. It must specify a one-character value. In the absence of a trim specifier, BOTH is assumed. In the absence of an explicit pad character, BLANK is assumed.
Parameters

*leading*
Indicates the orientation of the TRIM function.

*trailing*
Indicates the orientation of the TRIM function.

*both*
Indicates the orientation of the TRIM function.

**TRUNCATE-function**

Syntax

```
TRUNCATE ( value-expression1, value-expression2 )
```

Parameters

*value-expression1*
Specifies a numeric value-expression. See Expansion of Value-expression.

*value-expression2*
Specifies a numeric value-expression. See Expansion of Value-expression.

TRUNCATE returns value-expression1 truncated to value-expression2 places to the right of the decimal point if value-expression2 is positive or 0. If value-expression2 is negative, value-expression1 is truncated to the absolute value of value-expression2 places to the left of the decimal point. If the absolute value of value-expression2 is not smaller than the number of digits to the left of the decimal point, the result is 0.

value-expression1 must be of any numeric data type.

value-expression2 must be of any numeric data type but will be internally converted to INTEGER.

The result of the function has the same data type and attributes as value-expression1.

If any of the value-expressions are null, the result is the null value. If a data error occurs, an exception is raised.
Example

The following statement returns 627.46380 627.46300 627.46000 627.40000 627.00000 620.00000 600.00000 0.00000 0.00000:

```sql
SELECT TRUNCATE(627.46381, 4) ,
       TRUNCATE(627.46381, 3) ,
       TRUNCATE(627.46381, 2) ,
       TRUNCATE(627.46381, 1) ,
       TRUNCATE(627.46381, 0) ,
       TRUNCATE(627.46381, -1) ,
       TRUNCATE(627.46381, -2) ,
       TRUNCATE(627.46381, -3) ,
       TRUNCATE(627.46381, -4)
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**UCASE or UPPER-function**

Syntax

```
UCASE (or UPPER) operates on CHARACTER or VARCHAR value-expressions. The result is a string of equal length where all lower case characters have been folded into upper case.
```

**USER-function**

Syntax

```
USER is equivalent to the special-register USER. See Expansion of Special-register for more information.
```

Example

The following statement returns ABCDE01, user executing the SELECT statement:

```sql
SELECT USER()
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

**VALUE or COALESCE-function**

Syntax

```
VALUE (or COALESCE) function
```

The VALUE scalar function is used to substitute a value for the null value.

The data types of the value-expressions must be compatible. Character strings are not converted to date/time values. Therefore, if any value-expression is a date, each value-expression must be a date, if any value-expression is a time, each value-expression must be a time, if any value-expression is a timestamp, each value-expression must be a timestamp, and if any value-expression is a character string, each value-expression must be a character string.

The value-expressions are evaluated in the order they are specified, and the result of the function is equal to the first value-expression that is not null. The result can be null only if all value-expressions are null.

The result is defined as "equal to" a value-expression because that value-expression is converted or extended, if necessary, to conform to the data type of the function. The data type of the result is derived from the data types of the specified value-expressions as follows:

**Strings:** If any value-expression 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 value-expressions 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:** If the value-expressions are dates, the result is a date. If the value-expressions are times, the result is a time. If the value-expressions are timestamps, the result is a timestamp.

**Numbers:** If the value-expressions are numbers, the result is the numeric data type that would occur if all value-expressions were part of a single arithmetic expression. If that data type is decimal or numeric, it has precision of \( p \) and scale of \( s \) so \( s \) is the largest result scale of any value-expression, and \( p \) is \( s + n \), where \( n \) is the largest integral part of any value-expression. Conversion errors are possible if \( s + n \) is greater than 31.

**VARGRAPHIC-function**

**Syntax**

\[
\text{VARGRAPHIC ( value-expression )}
\]
The VARGRAPHIC function is supported only in active DBCS environments.

The VARGRAPHIC function is used to obtain a graphic string representation of a character string. **Value-expression** must be a binary or character string. If it is binary, the value is treated as graphic, and no data conversion takes place.

If the **value-expression** must be a character string, the characters are converted to their DBCS equivalent. If the string contains shift-in and shift-out characters, they must be properly paired under the rules for mixed data.

The result of the function is a varying length graphic string. If the **value-expression** can be null, the result can be null; if the **value-expression** is null, the result is the null value.

If **value-expression** is character, it is interpreted as a mixed data string. The result includes all DBCS characters of the **value-expression** and the DBCS equivalent of all single byte characters of the **value-expression**. The first character of the result is the first logical character of the **value-expression**, the second character of the result is the second logical character of the **value-expression**, and so forth. The result does not include shift-in and shift-out characters.

The length of the result depends on the number of logical characters in the **value-expression**. If the length or maximum length of the **value-expression** is \( n \) bytes, the maximum length of the result is \( n \) (DBCS characters).

### WEEK-function

**Syntax**

```sql
WEEK ( value-expression )
```

WEEK returns the week of the year for the specified **value-expression**. The function uses the ISO definition: a week starts with Monday and comprises 7 days. Week 1 is the first week of the year that contains a Thursday (or the first week that contains January 4).

**value-expression** must be a DATE or TIMESTAMP data type or must be a CHARACTER or VARCHAR data type and represent a valid string representation of a date or timestamp.

The result is an INTEGER data type and is in the range of 1 to 53.

The result is null if value-expression is null.

**Example**

The following statement returns 52 1:

```sql
SELECT WEEK ('2000-01-01'), WEEK('2000-01-03')
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM'
```
XMLPOINTER-function

Returns a BINARY(4) value that is a pointer to a LOB (Large Object) that holds the serialized value of XML-value-expression.

The XMLPOINTER function is used in programs that need to process serialized XML values. The structure of the LOB is a variable-length storage object. It starts with a signed integer of 32 bit and contains the LOB data length (max 2 GB), followed by the LOB data.

Syntax

```
XMLPOINTER (XML-value-expression)
```

Notes:

- If XML-value-expression is NULL or empty, XMLPOINTER returns a NULL value.
- The storage object of the LOB is allocated from a CA IDMS CV storage pool or from the batch address space for local mode programs. The storage object is only addressable in client programs that run in the same CA IDMS CV as the database server or in batch local mode programs.
- The program invoking the XMLPOINTER function must free the storage of the LOB when it is no longer needed. If no free storage is done, the storage associated with the LOB is freed at task termination.
- Client programs that cannot access the LOB returned by XMLPOINTER can use XMLSERIALIZE (returns a maximum of 30,000 characters) or the table procedure SYSCA.XMLSLICE to process XML values.
- XMLPOINTER is not part of the SQL standard. It is a CA IDMS extension to facilitate access to XML LOBs.

Example

The following statement returns pointers to XML LOB objects:

```
SELECT XMLPOINTER(XMLFOREST(NAME as "Name"
, SCHEMA as "Schema")
) AS "PointerToLob"
FROM  SYSTEM.TABLE
where schema = 'DEMOPROJ'
```

The result is similar to the following:

```
+------------------
| PointerToLob     |
| 2000300800       |
| 2000300800       |
```
XMLSERIALIZE-function

Returns a value of character string or binary string. Serialization is an operation on an XML value that transforms the XML value in a continuous character string representation. Serialization is the inverse operation of parsing.

Syntax

```sql
XMLSERIALIZE ( CONTENT DOCUMENT )
XML-value-expression AS string-data-type )
```

Parameters

`string-data-type`

Must be one of the character data types of data type: CHAR(n), CHARACTER(n), VARCHAR(n), CHAR VARYING(n).

**Note:** The DOCUMENT option is not functional in this feature. Therefore, CONTENT should always be specified.

Example

Use of XMLSERIALIZE to serialize an XML value as a character string of 50 characters.

```sql
SELECT NAME,
       XMLSERIALIZE(CONTENT
          XMLELEMENT(NAME "Schema"
            XMLATTRIBUTES (NAME AS "Name"
               ,USER AS "User"))
       AS CHAR(50)) AS "Serialized XML"
FROM SYSTEM.SCHEMA WHERE NAME IN ('SYSTEM', 'SYSDICT') ;
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>Serialized XML</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSTEM</td>
<td><code>&lt;Schema Name=&quot;SYSTEM&quot; User=&quot;ABCDE01&quot;&gt;&lt;/Schema&gt;</code></td>
</tr>
<tr>
<td>SYSDICT</td>
<td><code>&lt;Schema Name=&quot;SYSDICT&quot; User=&quot;VWXYZ01&quot;&gt;&lt;/Schema&gt;</code></td>
</tr>
</tbody>
</table>

YEAR-function

Syntax

```sql
YEAR ( value-expression )
```
YEAR obtains the year part of the value in value-expression. value-expression must be a date, timestamp, or date duration.

The result of the YEAR function is an integer, as shown in the following table.

<table>
<thead>
<tr>
<th>Value-expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP value</td>
<td>1 to 9999 (the year part of the timestamp)</td>
</tr>
<tr>
<td>DATE value</td>
<td>1 to 9999 (the year part of the date)</td>
</tr>
<tr>
<td>Date duration</td>
<td>The year part of the value (an integer in the range -9999 to 9999 with the same sign as value-expression if the result is not 0)</td>
</tr>
</tbody>
</table>

**Expansion of User-defined-function**

This section describes how user-defined functions are invoked, including the purpose, syntax, parameters, usage considerations, and examples.

A user-defined function is invoked through a qualified or unqualified function identifier together with an optional set of parameter values and returns a single value. To invoke a user-defined function, you must either own or hold the SELECT privilege on the named function.

**Syntax**

```
Expansion of user-defined-function

[schema-name.] function-identifier

( parameter-specification )

Expansion of parameter-specification

parameter-name = value-expression
```
Parameters

**schema-name**

Specifies the schema with which the function identified by function-identifier is associated.

**Note:** For more information about using a schema name to qualify a function, see Identifying Entities in Schemas, (see page 42)

**function-identifier**

Identifies a function defined in the dictionary.

**parameter-specification**

Specifies a value to be assigned to a parameter of a function. Both the positional (with NO parameter-name) and the non-positional (with parameter-name) forms of parameter specification can be used in a single function invocation. If a non-positional parameter specification is used, all remaining parameter specifications in the parameter list MUST be non-positional. Positional parameter specifications are assumed to correspond to the declared parameters of a function in the sequence of their declaration.

**parameter-name**

Specifies the name of a parameter associated with the function.

**value-expression**

Specifies a value-expression. See Expansion of Value-expression.

Usage

**Passing and Returning Values to a Function:** During SQL function processing, CA IDMS issues a call to the corresponding SQL-invoked routine with the values supplied in the function invocation. Before returning control, the SQL-invoked routine must set a value for the implicitly defined output parameter USER_FUNC; this then becomes the function return value.

**Usage Restriction:** You cannot reference a user-defined function within the search condition of a table's check constraint.

Examples

The invocation of the function UDF_FUNBOUS, defined in the schema FIN, causes the external program FUNBONUS to be called by CA IDMS with two parameters. The first parameter contains the value for EMP_ID, the second is the implicitly defined parameter USER_FUNC, which needs to be given a value by FUNBONUS before returning control to CA IDMS.

```sql
SELECT EMP_ID, FUN.UDF_FUNBOUS(EMP_ID) FROM DEMOEMPL.EMPLOYEE;
```
Expansion of XML-value-function

The expanded parameters of XML-value-function represent the invocation of an XML-value-function.

Syntax

Expansion of XML-value-function

Parameters

Specifies the XML-value-function to be invoked. For detailed descriptions of the XML value functions, see XML Value Functions.

XML Value Functions

This section describes the XML value functions including their purpose, syntax, parameters, and examples.

XMLAGG-function

Returns an XML value that is computed from a collection of rows. The result is the XML concatenation of a list of XML elements, aggregated in the statement containing the XMLAGG-function.
Parameters

ORDER BY

Before the aggregation takes place, the XML elements, specified by XML-value-expression, are sorted in ascending or descending order by the values in the specified columns. XML elements are ordered first by the first column specified, then by the second column specified within the ordering established by the first column, then by the third column specified, and so on.

col-nm

Specifies the name of column.

table-name.

Specifies the table, view, procedure, or table procedure that includes the named column. For expanded table-name syntax, see Expansion of Table-name.

alias

Specifies the alias associated with the table, view, procedure, or table procedure that includes the named column. The alias must be defined in the FROM parameter of the subquery, query specification, or SELECT statement that includes the XMLAGG function.

column-number

Specifies a column number. You can specify from 1 through 254 columns. Multiple columns must be separated by commas.
Examples

Example 1

Use of the XMLAGG function to display all employees belonging to each department.

```
SELECT XMLSERIALIZE(CONTENT
  XMLELEMENT(NAME "dept",
    XMLATTRIBUTES(e.DEPT_ID AS "id"),
    XMLAGG(XMLELEMENT(NAME "lname",
      e.EMP_LNAME))
  AS VARCHAR(256)) AS "EMP_NAME_COL"
FROM DEMOEMPL.EMPLOYEE e GROUP BY DEPT_ID ;
```

The result is similar to the following. Note that the content of the EMP_NAME_COL column has been formatted for convenience.

```
EMP_NAME_COL
-------------
<dept id="1100">
  <lname>Fordman</lname>
  <lname>Halloran</lname>
  <lname>Hamel</lname>
</dept>
<dept id="1110">
  <lname>Widman</lname>
  <lname>Alexander</lname>
</dept>
<dept id="1120">
  <lname>Umidy</lname>
  <lname>White</lname>
  <lname>Johnson</lname>
</dept>
```

Example 2

Use of the XMLAGG function to display all employees belonging to each department. For each employee, the positions and jobs are included. This example shows that the use of the XMLAGG function together with the ability to specify subqueries as arguments for the SQL/XML functions allows creating very complex XML structures.
select xmlpointer (xml
xmlelement
 ( Name "Employees"
 , xmlagg
  ( xmlelement
   ( NAME "Department"
    , xmlattributes(DEPT_ID as "DeptId"
    , xmlelement
     ( NAME "EmployeesInDepartment"
      , select xmlagg
       ( xmlelement
        ( name "Employee"
         , xmlattributes(EMP_ID as "EmpId"
         , EMP_FNAME
         , EMP_LNAME
         , xmlelement
          ( name "Address"
           , XMLFOREST
            ( e.STREET as "Street"
            , e.CITY as "City"
            , e.STATE as "State"
            )
            )
           
           )
          , xmlelement
           ( name "Positions"
            , select xmlagg
             ( xmlelement
              ( name "Position"
               , xmlattributes
                ( p.JOB_ID as "JobId" )
               , JOB_TITLE
               , SALARY_AMOUNT
               , BONUS_PERCENT
               )
               )
             
             )
           )
          from DEMOEMPL.POSITION p, DEMOEMPL.JOB j
          where p.EMP_ID = e.EMP_ID
          and p.JOB_ID = j.JOB_ID
         )
      )
   )
from DEMOEMPL.EMPLOYEE e
where d.DEPT_ID = e.DEPT_ID
)
)from DEMOEMPL.DEPARTMENT d
The result is similar to the following. It has been formatted and displayed with an "XML-enabled" Web browser that allows collapsing and expanding XML elements in an XML tree.

- <Employees>
  + <Department DeptId="1120">
  - <Department DeptId="5000">
  - <EmployeesInDepartment>
  - <Employee EmpId="3449">
    Cynthia Taylor
  - <Address>
    <Street>201 Washington St</Street>
    <City>Concord</City>
    <State>MA</State>
  - <Positions>
    <Position JobId="4023">Accountant 74776.0</Position>
  </Positions>
  </Employee>
  + <Employee EmpId="5103">
    &invellip.
  </EmployeesInDepartment>
  </Department>
+ <Department DeptId="4500">
</Employees>
Example 3

Use of the XMLAGG function and subqueries to display part of an SQL catalog as an XML document.

```sql
select xmlpointer ( 
  xmlelement
  ( Name "Catalog"
    , xmlagg
    ( xmlelement
      ( Name "Schema"
        , xmlattributes
          ( s.NAME as "Name"
            , s.TYPE as "Type"
          )
        , 'Referencing SQL Schema:'
        , s.REFDSQLSCHEMA
        , 'Referencing Non SQL Schema:'
        , s.NTWKSCHEMA
        , select
          xmlagg
          ( xmlelement
            ( Name "TablesInSchema"
              , xmlattributes
                ( t.NAME as "Name"
                  , t.TYPE as "Type"
                )
            , SEGMENT
            , '.
            , 'AREA'
            , xmlelement
              ( Name "TableStats"
                , xmlattributes
                  ( t.NUMCOLS as "NumCols"
                    , t.NUMINDEXES as "NumIndexes"
                    , t.NUMREFERENCING as "NumReferencing"
                    , t.NUMROWS as "NumRows"
                    , t.NUMPAGES as "NumPages"
                    , t.NUMSYNTAX as "NumSyntax"
                    , t.ESTROWS as "EstRows"
                  )
                )
            , select
              xmlagg
              ( xmlelement
                ( Name "ColumnsInTable"
                  , xmlattributes
                    ( c.NAME as "Name"
                      , c.NUMBER as "Nr"
                    )
                )
            )
          )
    )
  )
)
XML Value Functions

Chapter 5: Functions

```sql
xml

FROM SYSTEM.COLUMN c
WHERE c.TABLE = t.NAME
    AND c.SCHEMA = t.SCHEMA

FROM SYSTEM.TABLE t
WHERE t.SCHEMA = s.NAME
    AND TYPE = 'T'

FROM SYSTEM.SCHEMA s
```
The result is similar to the following. It has been formatted and displayed with an "XML-enabled" Web browser that allows collapsing and expanding XML elements in an XML tree.

```xml
- <Catalog>
  <Schema Name="EMPSCHM" Type="N">  
    Referencing SQL Schema: Referencing Non SQL Schema:EMPSCHM</Schema>
- <Schema Name="DEMOEMPL" Type="R">  
    Referencing SQL Schema:Referencing Non SQL Schema:
+ <TablesInSchema Name="DEPARTMENT" Type="T" Length="68">  
+ <TablesInSchema Name="DIVISION" Type="T" Length="56">  
+ <TablesInSchema Name="EMPL_MANAGER_INFO" Type="T" Length="56">  
+ <TablesInSchema Name="EMPLOYEE" Type="T" Length="204">  
  SQLDEMO .AREA
  <TableStats NumCols="15" NumIndexes="4" NumReferencing="2" NumRows="55"
    NumPages="40" NumSyntax="1" EstRows="0" />
- <ColumnsInTable Name="DEPT_ID" Nr="5">  
  UNSIGNED NUMERIC
  <DataTypeDetails Code="128" Precision="4" Scale="0" />
  <OtherDetails Null="N" Default="N" VOffset="49" VLength="4" NOffset="0"
    NLength="0" NumValues="14" />
</ColumnsInTable>
- <ColumnsInTable Name="EMP_FNAME" Nr="3">  
  CHARACTER
  <DataTypeDetails Code="1" Precision="0" Scale="0" />
  <OtherDetails Null="N" Default="N" VOffset="9" VLength="20" NOffset="0"
    NLength="0" NumValues="0" />
</ColumnsInTable>
- <ColumnsInTable Name="EMP_ID" Nr="1">  
  UNSIGNED NUMERIC
  <DataTypeDetails Code="128" Precision="4" Scale="0" />
  <OtherDetails Null="N" Default="N" VOffset="0" VLength="4" NOffset="0"
    NLength="0" NumValues="0" />
</ColumnsInTable>
  &invellip.
</TablesInSchema>
  &invellip.
+ <TablesInSchema Name="INSURANCE_PLAN" Type="T" Length="168">  
+ <TablesInSchema Name="JOB" Type="T" Length="188">  
+ <TablesInSchema Name="POSITION" Type="T" Length="64">  
</Schema>
  &invellip.
</Catalog>
```
XMLCOMMENT-function

Returns an XML value that is an XML comment, generated from string-value-expression. The XML value consists of an XML root information item with one child, an XML comment information item whose [content] property is string-value-expression.

Syntax

XMLCOMMENT ( string-value-expression )

Parameters

string-value-expression

Specifies a character string value-expression, that is a value-expression that returns a value of type character.

If string-value-expression is NULL, XMLCOMMENT returns a NULL value. string-value-expression cannot contain a hyphen (--) sequence of characters and cannot end with a hyphen (-) character.

Example

The following statement returns a single XML comment:

```sql
SELECT XMLSERIALIZE(CONTENT XMLCOMMENT('My personal opinion')
    AS CHAR(80)) as "Comment Only"
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

The result is similar to the following:

Comment Only

-------------

<!--My personal opinion-->
Example

Use of the XMLCONCAT function to concatenate two XML elements defined using the XMLELEMENT function.

```
SELECT e.EMP_ID,
    XMLSERIALIZE(CONTENT
        XMLCONCAT(XMLELEMENT(NAME "fname",
            e.EMP_FNAME),
        XMLELEMENT(NAME "lname",
            e.EMP_LNAME))
    AS VARCHAR(64)) AS "EMP_NAME_COL"
FROM DEMOEMPL.EMPLOYEE e WHERE EMP_ID < 1500 ;
```

The result is similar to the following:

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>EMP_NAME_COL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1003</td>
<td>&lt;fname&gt;James&lt;/fname&gt;&lt;lname&gt;Baldwin&lt;/lname&gt;</td>
</tr>
<tr>
<td>1034</td>
<td>&lt;fname&gt;James&lt;/fname&gt;&lt;lname&gt;Gallway&lt;/lname&gt;</td>
</tr>
<tr>
<td>1234</td>
<td>&lt;fname&gt;Thomas&lt;/fname&gt;&lt;lname&gt;Mills&lt;/lname&gt;</td>
</tr>
</tbody>
</table>

XMLELEMENT-function

Returns an XML value that is a single XML element information item as a child of its XML root information. Provided are an XML element name, an optional list of XML namespace declarations, an optional list of attributes, and an optional list of values as the content of the new element.

The XMLATTRIBUTES pseudo function can be used to specify XML attributes in an XML element. The XMLNAMESPACES pseudo function can be used to declare XML namespace in an XML element.

Syntax

```
XMLELEMENT ──(NAME ─ XML-element-name )

[ . XML-namespace-declaration ] [ . XML-attributes ]

└─ ▼ XML-content-val-exp ─ OPTION ─ NULL ON NULL ─ EMPTY ON NULL ─ ABSENT ON NULL ─ NIL ON NULL ─ NIL ON NO CONTENT

Expansion of XML-namespace-declaration

XMLNAMESPACES ──(─▼─ XML-namespace-declaration-item ─┐

```

```
XML ELEMENT ──(NAME ─ XML-element-name )

[ . XML-namespace-declaration ] [ . XML-attributes ]

└─ ▼ XML-content-val-exp ─ OPTION ─ NULL ON NULL ─ EMPTY ON NULL ─ ABSENT ON NULL ─ NIL ON NULL ─ NIL ON NO CONTENT

Expansion of XML-namespace-declaration

XMLNAMESPACES ──(─▼─ XML-namespace-declaration-item ─┐

```

```
XML ELEMENT ──(NAME ─ XML-element-name )

[ . XML-namespace-declaration ] [ . XML-attributes ]

└─ ▼ XML-content-val-exp ─ OPTION ─ NULL ON NULL ─ EMPTY ON NULL ─ ABSENT ON NULL ─ NIL ON NULL ─ NIL ON NO CONTENT

Expansion of XML-namespace-declaration

XMLNAMESPACES ──(─▼─ XML-namespace-declaration-item ─┐

```
Expansion of XML-namespace-declaration-item

XML-namespace-URI-char-lit AS XML-namespace-prefix-id

DEFAULT ─ XML-namespace-URI-char-lit

NO DEFAULT

Expansion of XML-attributes

XMLATTRIBUTES ─(─▼─ XML-att-val-exp ─ AS XML-att-name)─►◄

Parameters

XML-element-name

Specifies an identifier that is used as an XML element name. This name must be an XML QName. If the name is qualified, the namespace prefix must be declared within the scope. The maximum length of the identifier is 128 characters.

XML-content-val-exp

Specifies a value-expression or an XML-value-expression that after mapping according to Mapping SQL Data Type Values to XML Schema Data Type Values, is used as the content of the generated XML element.

XML-namespace-URI-char-lit

Specifies a character string literal of an XML namespace through a URI. For example, http://www.w3.org/2001/XMLSchema. This character string literal can be empty when used with the DEFAULT option only.
XML namespace-prefix-id

Specifies an identifier that is used as a namespace prefix that is bound to the XML namespace given by XML-namespace-uri-char-lit. The maximum length of the identifier is 128 characters.

This identifier must be an XML NCName. It cannot be equal to "xml" or "xmlns", and it cannot start with the characters "xml" (in any combination). Be sure that no duplicate namespace prefixes are declared in the same XMLNAMESPACES function call.

XML-att-name

Specifies an identifier that is used as the XML attribute name. The maximum length of the identifier is 128 characters. The attribute name must be an XML QName. It cannot be equal to "xmlns" or start with "xmlns:". Be sure that no duplicate attribute names are declared in the same XMLATTRIBUTES function call.

XML-att-val-exp

Specifies a value-expression that is used as the value of the XML attribute. The value-expression can be of any type except GRAPHIC or VARGRAPHIC. The length of the value is limited to 512 characters.

If XML-att-name is not specified, the attribute name is derived from the XML-att-val-exp value. The XML-att-val-exp value must be a valid SQL column name, optionally qualified with table-name or alias. The fully escaped mapping is applied on the SQL column name to create the attribute name.

OPTION

Specifies the processing of null values for XML-content-val-exp as follows:

ABSENT ON NULL

The returned value is never null, but element is completely absent when all XML-content-val-exp are NULL.

EMPTY ON NULL

The returned value is never null. Element has no content for each NULL value of XML-content-val-exp that is NULL. This is the default.

NIL ON NO CONTENT

The returned value is not null. When the [children] property of the element does not contain at least one XML element information item or at least one XML character information item, the element is:

<XML-element-name xsi:nil="true"/>
NIL ON NULL

The returned value is not null, but when all XML-content-val-exp are NULL, element is

<XML-element-name xsi:nil="true"/>


NULL ON NULL

The returned value is NULL when all XML-content-val-exp are NULL.

Notes:

■ XMLATTRIBUTES look like any other SQL/XML function, but it is not an SQL function. It is a function-like construct that can only be used in an XMLELEMENT invocation.

■ An XML-namespace-declaration can have only one XML-namespace-declaration-item containing the literal DEFAULT or NO DEFAULT.

■ XML-element-name is an identifier in the SQL language. Some valid XML names, that is, all XML QNames with a non-null prefix, require this identifier to be delimited by double quotes.

Examples

Example 1

The following SELECT statement produces a row for each employee with one column representing an 'emp' XML element containing the employee’s last name. The data type of the result column is VARCHAR(64).

SELECT XMLSERIALIZE(CONTENT
  XMLELEMENT(NAME "emp", EMP_LNAME)
    AS VARCHAR(64)) AS "EMP_NAME.Col"
FROM DEMOEMPL.EMPLOYEE ;

The result is similar to the following:

EMP_NAME.Col
-----------
<emp>Baldwin</emp>
<emp>Gallway</emp>
<emp>Mills</emp>
Example 2

Same as Example 1, but this example includes a first column containing the employee ID, which uses the "e" alias for the table name and a WHERE clause on the SELECT statement.

SELECT e.EMP_ID,
       XMLSERIALIZE(CONTENT
                        XMLELEMENT(NAME "emp", e.EMP_LNAME)
                        AS VARCHAR(64)) AS "EMP_NAME_COL"
FROM DEMOEMPL.EMPLOYEE e WHERE EMP_ID < 1500 ;

The result is similar to the following:

EMP_ID  EMP_NAME_COL
------  ---------------
 1003   <emp>Baldwin</emp>
 1034   <emp>Gallway</emp>
 1234   <emp>Mills</emp>

Example 3

Same as Example 2, but this example also includes the employee ID as an attribute within the <emp> tag.

SELECT e.EMP_ID,
       XMLSERIALIZE(CONTENT
                        XMLELEMENT(NAME "emp",
                                      XMLATTRIBUTES(e.EMP_ID AS "id"),
                                      e.EMP_LNAME)
                        AS VARCHAR(64)) AS "EMP_NAME_COL"
FROM DEMOEMPL.EMPLOYEE e WHERE EMP_ID < 1500 ;

The result is similar to the following:

EMP_ID  EMP_NAME_COL
------  ---------------
 1003   <emp id="1003">Baldwin</emp>
 1034   <emp id="1034">Gallway</emp>
 1234   <emp id="1234">Mills</emp>
Example 4

Same as Example 3, but this example includes a second attribute within the `<emp>` tag with the employee's first name.

```
SELECT e.EMP_ID,
    XMLSERIALIZE(CONTENT
        XMLELEMENT(NAME "emp",
            XMLATTRIBUTES(e.EMP_ID AS "id",
                           e.EMP_FNAME AS "fname"),
                        e.EMP_LNAME)
        AS VARCHAR(64)) AS "EMP_NAME_COL"
FROM DEMOEMPL.EMPLOYEE e WHERE EMP_ID < 1500 ;
```

The result is similar to the following. Note that the content of the `EMP_NAME_COL` column has been formatted for convenience.

```
EMP_ID  EMP_NAME_COL
------  -------------------
1003    <emp id="1003" fname="James">Baldwin</emp>
1034    <emp id="1034" fname="James">Gallway</emp>
1234    <emp id="1234" fname="Thomas">Mills</emp>
```
Example 5

Same as Example 4, but this example removes the attributes and uses the employee's first name and last name as sub-elements of <emp>.

```
SELECT e.EMP_ID,
       XMLSERIALIZE(CONTENT
           XMLELEMENT(NAME "emp",
                         XMLELEMENT(NAME "fname", e.EMP_FNAME),
                         XMLELEMENT(NAME "lname", e.EMP_LNAME))
       AS VARCHAR(64)) AS "EMP_NAME_COL"
FROM DEMOEMPL.EMPLOYEE e WHERE EMP_ID < 1500 ;
```

The result is similar to the following. Note that the content of the EMP_NAME_COL column has been formatted for convenience.

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>EMP_NAME_COL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1003</td>
<td>&lt;emp&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;fname&gt;James&lt;/fname&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;lname&gt;Baldwin&lt;/lname&gt;</td>
</tr>
<tr>
<td>1034</td>
<td>&lt;emp&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;fname&gt;James&lt;/fname&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;lname&gt;Gallway&lt;/lname&gt;</td>
</tr>
<tr>
<td>1234</td>
<td>&lt;emp&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;fname&gt;Thomas&lt;/fname&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;lname&gt;Mills&lt;/lname&gt;</td>
</tr>
</tbody>
</table>

Example 6

Same as Example 5, but this example concatenates the employee's first name and last name into one sub-element of <emp>.

```
SELECT e.EMP_ID,
    XMLSERIALIZE(CONTENT
        XMLELEMENT(NAME "emp",
            XMLELEMENT(NAME "name",
                e.EMP_FNAME || ' ' || e.EMP_LNAME)
            AS VARCHAR(64)) AS "EMP_NAME_COL"
FROM DEMOEMPL.EMPLOYEE e WHERE EMP_ID < 1500 ;
```

The result is similar to the following. Note that the content of the EMP_NAME_COL column has been formatted for convenience.

```
EMP_ID  EMP_NAME_COL
------  ----------------
1003    <emp>
       <name>James Baldwin</name>
       </emp>
1034    <emp>
       <name>James Gallway</name>
       </emp>
1234    <emp>
       <name>Thomas Mills</name>
       </emp>
```
Example 7

Same as Example 6, but this example uses XMLNAMESPACES.

```sql
SELECT e.EMP_ID,
       XMLSERIALIZE(CONTENT
                      XMLELEMENT(NAME "emp",
                                  XMLNAMESPACES(
                                      DEFAULT 'http://ca.com/hr/globalxml',
                                      'http://ca.com/hr/frenchxml' AS "fr" ),
                                  XMLELEMENT(NAME "fr:nom",
                                              e.EMP_FNAME || ' ' || e.EMP_LNAME)
                                 )
                      ) AS "EMP_NAME_COL"
FROM DEMOEMPL.EMPLOYEE e WHERE EMP_ID < 1500 ;
```

The result is similar to the following. Note that the content of the EMP_NAME_COL column has been formatted for convenience.

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>EMP_NAME_COL</th>
</tr>
</thead>
</table>
| 1003   | <emp xmlns="http://ca.com/hr/globalxml"
            xmlns:fr="http://ca.com/hr/frenchxml">
            <fr:nom>James Baldwin</fr:nom>
        </emp> |
| 1034   | <emp xmlns="http://ca.com/hr/globalxml"
            xmlns:fr="http://ca.com/hr/frenchxml">
            <fr:nom>James Gallway</fr:nom>
        </emp> |
| 1234   | <emp xmlns="http://ca.com/hr/globalxml"
            xmlns:fr="http://ca.com/hr/frenchxml">
            <fr:nom>Thomas Mills</fr:nom>
        </emp> |
Example 8

This example illustrates the use of a subquery as an argument of XMLELEMENT.

```sql
SELECT XMLSERIALIZE
    ( CONTENT
        XMLELEMENT
            ( NAME "Employee"
                ,XMLATTRIBUTES(e.EMP_ID as "Id")
                , e.EMP_FNAME
                , e.EMP_LNAME
                , SELECT
                    XMLELEMENT
                        ( NAME "Manager"
                            , XMLATTRIBUTES(m.EMP_ID as "MgrId")
                            , m.EMP_FNAME || m.EMP_LNAME
                        )
                    FROM DEMOEMPL.employee m
                    WHERE e.MANAGER_ID = m.EMP_ID
                )
         ) AS VARCHAR(120)) AS "EmployeeManager"
FROM DEMOEMPL.EMPLOYEE e
```

The result is similar to the following:

```
EmployeeManager
--------------
<Employee Id="5008">Timothy Fordman
    <Manager MgrId="2246">Marylou Hamel</Manager></Employee>
<Employee Id="4703">Martin Halloran
    <Manager MgrId="2246">Marylou Hamel</Manager></Employee>
```
XMLFOREST-function

Returns an XML value that is a list of XML element information items as the children of its XML root information. An XML element is produced from each XML-forest-val-exp, using the column name or, if provided, the XML-forest-elem-ident as the XML element name and the XML-forest-val-exp as the element content. The value of XML-forest-val-exp can be any value that has a mapping to an XML value.

The XMLNAMESPACES pseudo function can be used to declare XML namespace in an XML element.

Syntax

```
XMLFOREST ( XML-namespace-declaration ,
            XML-forest-val-exp AS XML-forest-elem-ident )
```

Note: For more information about the expansion of XML-namespace-declaration, see XMLELEMENT-function.

Parameters

**XML-forest-elem-ident**

Specifies an identifier that is used as an XML element name. The identifier must be an XML QName. If a namespace prefix is used, it must have been declared in the scope of the element. The maximum length of the identifier is 128 characters.

**XML-forest-val-exp**

Specifies a value-expression that is used as the element content of an XML element. If XML-forest-elem-ident is not specified, the forest element name is derived from the XML-forest-val-exp value. The XML-forest-val-exp value must be a valid SQL column name, optionally qualified with table-name or alias. The fully escaped mapping is applied on the SQL column name to create the forest element name.
OPTION

Specifies the processing of null values for $XML$-$forest$-$val$-$exp$ as follows:

**ABSENT ON NULL**

The returned value is never null, but element is completely absent from the list when $XML$-$forest$-$val$-$exp$ is NULL.

**EMPTY ON NULL**

The returned value is never null. Element has no content for each NULL value of $XML$-$forest$-$val$-$exp$.

**NIL ON NO CONTENT**

The returned value is not null. When the [children] property of element does not contain at least one XML element information item or at least one XML character information item, the element is:

```xml
<XML-forest-element-name xsi:nil="true"/>
```

**NIL ON NULL**

The returned value is not null, but when an $XML$-$forest$-$val$-$exp$ is NULL, element becomes

```xml
<XML-forest-element-name xsi:nil="true"/>
```


**NULL ON NULL**

The returned value is NULL when all $XML$-$forest$-$val$-$exp$ are NULL. This is the default.

**Note:** $XML$-element-name is an identifier in the SQL language. Some valid XML names, that is, all XML QNames with non-null namespace prefix, requires this identifier to be delimited by double quotes.
XML Value Functions

Example

Similar to Example 5 of the XMLELEMENT function, but the use of two XMLELEMENT invocations to declare two sub-elements of <emp> is replaced by a single XMLFOREST invocation.

```
SELECT e.EMP_ID,
    XMLSERIALIZE(CONTENT
        XMLELEMENT(NAM
            XMLFOREST(e.EMP_FNAME AS "fname",
                e.EMP_LNAME AS "lname")
            AS VARCHAR(64)) AS "EMP_NAME_COL"
FROM DEMOEMPL.EMPLOYEE e WHERE EMP_ID < 1500 ;
```

The result is similar to the following. Note that the content of the EMP_NAME_COL column has been formatted for convenience.

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>EMP_NAME_COL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1003</td>
<td>&lt;emp&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;fname&gt;James&lt;/fname&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;lname&gt;Baldwin&lt;/lname&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/emp&gt;</td>
</tr>
<tr>
<td>1034</td>
<td>&lt;emp&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;fname&gt;James&lt;/fname&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;lname&gt;Gallway&lt;/lname&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/emp&gt;</td>
</tr>
<tr>
<td>1234</td>
<td>&lt;emp&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;fname&gt;Thomas&lt;/fname&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;lname&gt;Mills&lt;/lname&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/emp&gt;</td>
</tr>
</tbody>
</table>

XMLPARSE-function

Returns an XML value as the result of performing a non-validating parse of a character string. Parsing is the inverse operation of serializing.

Syntax

```
XMLPARSE ─ ( CONTENT DOCUMENT string-value-expression )
   ──── STRIP PRESERVE WHITESPACE ──
```

Parameters

*string-value-expression*

Specifies a character string value-expression, that is a value-expression that returns a value of type character. If string-value-expression is NULL, XMLPARSE returns a NULL value.

**Note:** The DOCUMENT and STRIP WHITESPACE options are not functional in this feature. Therefore, CONTENT and PRESERVE WHITESPACE should always be specified.

**Example**

The following statement causes an SQL statement exception because the XML is not completely serialized. The serialization is truncated after 20 characters.

```sql
SELECT XMLPARSE ( CONTENT XMLSERIALIZE ( CONTENT XMLELEMENT ( NAME "EMP", EMP_LNAME ) AS CHAR(20) ) ) FROM DEMOEMPL.EMPLOYEE ;
```

*+ Status = -4       SQLSTATE = 38000       Messages follow:
*+ DB001075 C-4M321: Procedure IDMSQFUX exception 38000 XMLPARSE: Premature end of data in tag EMP line 1

**XMLPI-function**

Returns an XML value that is an XML processing instruction (PI). The XML value consists of:

- An XML root information item with one child
- An XML processing information item whose [target] property is the partially escaped mapping of identifier to an XML Name, and whose [content] property is string-value-expression, trimmed of leading blanks.

**Syntax**

```sql
XMLPI — ( NAME — identifier [ string-value-expression ] ) ➔
```
Parameters

**Identifier**

Specifies the target in the processing instruction. It must be a valid NCName. The maximum length of the *identifier* is 128 characters.

**string-value-expression**

Specifies a character string *value-expression*, that is a *value-expression* that returns a value of type character. It can be NULL or empty, but if present, it cannot contain the "?>" sequence.

A processing instruction takes the following syntactical form in XML 1.0:

```xml
<?target data?>
```

Processing instructions instruct applications to perform some type of extra processing on a given document.

An example of a processing instruction, which is supported by most Web browsers is:

```xml
<?xml-stylesheet href="mystyle.xsl" type="text/xsl"?>
```

When the browser loads an XML document and recognizes the processing instruction, it performs a transformation using the specified XSLT file and displays the result of the transformation instead of the raw XML file. This processing instruction has been accepted as a W3C recommendation. For more information, see http://www.w3.org/TR/xml-stylesheet.

Another example of a processing instruction accepted as a W3C recommendation is the use of the XML declaration:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

The pseudo-attribute *version* currently must have value "1.0". The pseudo-attribute *standalone* specifies whether any markup declarations are defined in separate documents. Finally, the pseudo-attribute *encoding* specifies the encoding of the XML document. XML parsers are required to support at least encoding UTF-8 and UTF-16.
Example

The following statement returns only a single processing instruction:

```
SELECT XMLSERIALIZE(CONTENT XMLPI("xml",
   'version="1.0" encoding="UTF-8" standalone="yes"')
AS CHAR(80)) as "PI Instruction"
FROM SYSTEM.SCHEMA WHERE NAME = 'SYSTEM';
```

The result is similar to the following:

```
PI Instruction
--------------
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

**XMLROOT-function**

Returns an XML value by modifying the properties of the XML root information item of another XML value.

All XML documents must have at least one well-formed root element. The root element, often called the document tag, must follow the prolog (XML declaration plus DTD) and must be a non-empty tag that encompasses the entire document.

**Notes:**

- The Encoding property cannot be specified in the XMLROOT function. However, the XMLROOT function sets the value of the property by using the value from the XML ENCODING parameter in the SQL SET SESSION statement, unless the given XML value already has a non-null value for its Encoding property.
- If the input **XML-value-expression** is NULL or empty, XMLROOT returns a NULL value.

**Syntax**

```
XMLROOT ( XML-value-expression
      , VERSION string-value-expression
      , STANDALONE YES)
```

**Parameters**

**string-value-expression**

Specifies a character string **value-expression**, that is a **value-expression** that returns a value of type character.
Example

Use of the XMLROOT function to generate the XML declaration in an XML document.

```sql
set session XML ENCODING UTF8;

select
    XMLSERIALIZE(CONTENT
        XMLROOT(
            XMLELEMENT(NAME "Employee",
                XMLATTRIBUTES(EMP_ID AS "Id",
                                DEPT_ID AS "DeptId",
                                MANAGER_ID AS "MgrId"),
                TRIM(EMP_FNAME)||' '||trim(EMP_LNAME),
                ' from ', CITY),
            VERSION '1.0', STANDALONE YES)
    AS VARCHAR(256)) AS "EmployeeData"
from DEMOEMPL.EMPLOYEE where EMP_ID = 1003;
```

The result is similar to the following. Note that the content of the EmployeeData column has been formatted for convenience:

```plaintext
EmployeeData
 -------------
 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Employee Id="1003" DeptId="6200">James Baldwin from Boston</Employee>
```
Chapter 6: Predicates and Search Condition

This section contains the following topics:

Overview (see page 207)
Expansion of Between-predicate (see page 207)
Expansion of Comparison-predicate (see page 209)
Expansion of Exists-predicate (see page 210)
Expansion of In-predicate (see page 211)
Expansion of Like-predicate (see page 214)
Expansion of Null-predicate (see page 219)
Expansion of Quantified-predicate (see page 220)
Expansion of Search-condition (see page 223)

Overview

This chapter presents expanded syntax for each predicate that can be used in a search condition. It concludes with expanded syntax for search-condition.

What is a Predicate?

A predicate is an operand of a search condition. It expresses or implies a comparison operation.

What is a Search Condition?

A search condition is a Boolean expression that yields a truth value. The operands of a search condition are predicates, and the operators are the logical operators AND, OR, and NOT.

A search condition establishes a criterion for selecting rows from a table.

More information:

ALTER TABLE (see page 290)
CREATE TABLE (see page 378)
Expansion of Search-condition (see page 223)

Expansion of Between-predicate

The between-predicate tests whether a value is within a range of values.
Expansion of Between-predicate

Usage

Comparable Data Types

The data types of the values specified in a between-predicate must be comparable.

Note: For information about comparing values of different data types, see Comparison, Assignment, Arithmetic, and Concatenation Operations. (see page 66)

Truth Value of a BETWEEN Predicate without NOT

The result of a BETWEEN predicate that does not include NOT is:

- True when value being tested is greater than or equal to the starting value of the test range and less than or equal to the ending value of the test range
- False when the value being tested is less than the starting value of the test range or greater than the ending value of the test range
- Unknown when one or more of the values specified in the predicate are null

Truth Value of a BETWEEN predicate with NOT

The result of a BETWEEN predicate that includes NOT is:

- True when value being tested is less than the starting value of the test range or greater than the ending value of the test range
- False when the value being tested is greater than or equal to the starting value of the test range and less than or equal to the ending value of the test range
- Unknown when one or more of the values specified in the predicate are null

Rounded Values

The BETWEEN predicate is useful for testing values derived from value expressions that may have been affected by rounding.

Example

As the Search Condition in a CHECK Parameter

The following CREATE TABLE statement defines the EXPERTISE table with four columns. The CHECK parameter defines a restriction on the data that can be stored in the SKILL_LEVEL column.

create table expertise
  (emp_id integer not null,
   skill_id integer not null,
   skill_level character(2),
   exp_date date,
   check (skill_level between '01' and '04');
More Information

- For more information about search conditions, see Expansion of Search-condition.
- For more information about the CHECK parameter, see CREATE TABLE.
- For more information about the ADD CHECK parameter, see ALTER TABLE.

Expansion of Comparison-predicate

The comparison-predicate tests whether a value is less than, equal to, or greater than another value.

Syntax

```
value-expression  comparison-operator  value-expression  ( subquery )
```

Parameters

**value-expression**

Specifies a value to be used in the comparison. For expanded `value-expression` syntax, see Expansion of Value-expression.

**comparison-operator**

Specifies the comparison operator to be used in the test. Valid values for `comparison-operator` are:

<table>
<thead>
<tr>
<th>Comparison operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
</tr>
<tr>
<td>-=</td>
<td>Not equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>=&gt;</td>
<td>Greater than or equal to</td>
</tr>
</tbody>
</table>
(subquery)

Specifies a subquery that returns no more than one row and whose result table consists of a single column.

**Note:** For more information about expanded subquery syntax, see [Expansion of Subquery](#) (see page 239)

## Usage

### Comparable Data Types

The data types of the values being compared must be comparable.

**Note:** For information about comparing values of different data types, see [Comparison, Assignment, Arithmetic, and Concatenation Operations](#) (see page 66)

### Truth Value of a Comparison Predicate

The result of a comparison predicate is:

- True when the first value relates to the second value in the way specified by the comparison operator
- False when the first value does not relate to the second value in the way specified by the comparison operator
- Unknown when one or both of the values being compared are null or when the result of the subquery is an empty set

## Example

### As the Search Condition in a WHERE Parameter

The following DELETE statement deletes rows in the EXPERTISE table for employees who were terminated before January 1, 2006. The search condition in the WHERE parameter of the subquery consists of a single comparison predicate.

```sql
delete from expertise
where emp_id in
    (select emp_id
     from employee
     where termination_date < '2006-01-01');
```

## Expansion of Exists-predicate

The exists-predicate tests for the existence of data meeting criteria specified in a subquery.
**Syntax**

 Expansion of exists-predicate

```
EXISTS ( subquery )
```

**Parameters**

( subquery )

Specifies a subquery that returns a set of zero or more rows. For expanded
subquery syntax, see Expansion of Subquery.

**Usage**

**Truth Value of an EXISTS Predicate**

The result of an EXISTS predicate is:

- True when the result of the subquery is a set containing one or more rows
- False when the result of the subquery is an empty set

**Example**

**With the Unary Operator NOT in a WHERE Parameter**

The following SELECT statement identifies employees for whom no expertise
information has been stored in the database.

```
select e1.emp_id
from employee e1
where not exists
  (select * from expertise e2
   where e1.emp_id = e2.emp_id);
```

**Note:** For more information about search conditions, see Expansion of
Search-condition.

**More information:**

Expansion of Search-condition (see page 223)

---

**Expansion of In-predicate**

The in-predicate tests whether a value occurs in a specified set of values.
Expansion of In-predicate

**Syntax**

```
Expansion of in-predicate

value-expression  NOT  IN

( value-expression subquery )
```

**Parameters**

- **value-expression**
  Specifies the value to be compared to the set of values identified by the **IN** parameter. For expanded **value-expression** syntax, see Expansion of **Value-expression**.

- **NOT**
  Reverses the test. **NOT** directs CA IDMS to test whether a value is not in the specified set of values.

- **IN**
  Identifies the set of values to which the value being tested is compared.

- **value-expression**
  Specifies a value that is a member of the set of test values. **Value-expression** may be enclosed in parentheses. Multiple occurrences of **value-expression** must be separated by commas and enclosed in parentheses.

- **subquery**
  Specifies a subquery that returns zero or more rows and whose result table consists of a single column. The column values are members of the set of test values. For expanded **subquery** syntax, see Expansion of Subquery.
Usage

Equivalence

Value-expression IN value-expression is equivalent to a comparison predicate in the form value-expression = value-expression.

Value-expression IN (subquery) is equivalent to a quantified predicate in the form value-expression = ANY (subquery).

Comparable Data Types

The data types of the values in an IN predicate must be comparable.

Note: For more information about comparing values of different data types, see Comparison, Assignment, Arithmetic, and Concatenation Operations. (see page 66)

Truth Value of an IN Predicate without NOT

The result of an IN predicate that does not include NOT is:
- True when the value being tested is equal to at least one of the values in the test set
- False when the value being tested is not equal to any of the values in the test set or when the result of the subquery is an empty set
- Unknown when the value being tested is null or when values in the test set are a combination of null value and values not equal to the value being tested

This table presents examples of results of IN predicates without NOT:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A' IN ('A','B')</td>
<td>True</td>
</tr>
<tr>
<td>'A' IN ('B')</td>
<td>False</td>
</tr>
<tr>
<td>'A' IN (null-value)</td>
<td>Unknown</td>
</tr>
<tr>
<td>'A' IN ('A',null-value)</td>
<td>True</td>
</tr>
<tr>
<td>'A' IN ('B',null-value)</td>
<td>Unknown</td>
</tr>
</tbody>
</table>
Truth Value of an IN Predicate with NOT

The result of an IN predicate that includes NOT is:
- True when the value being tested is not equal to any of the values in the test set or when the result of the subquery is an empty set
- False when the value being tested is equal to at least one of the values in the test set
- Unknown when the value being tested is null or values in the test set are a combination of null value and values not equal to the value being tested

This table presents examples of results of IN predicates with NOT:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A' NOT IN ('A','B')</td>
<td>False</td>
</tr>
<tr>
<td>'A' NOT IN ('B')</td>
<td>True</td>
</tr>
<tr>
<td>'A' NOT IN (null-value)</td>
<td>Unknown</td>
</tr>
<tr>
<td>'A' NOT IN ('A',null-value)</td>
<td>False</td>
</tr>
<tr>
<td>'A' NOT IN ('B',null-value)</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

Example

As the Search Condition in a WHERE Parameter

The following SELECT statement identifies employees who live in one of four specified cities:

```sql
select emp_fname, emp_lname, dept_id
from employee
where emp_city in ('Newton','Wellesley','Natick','Wayland');
```

Expansion of Like-predicate

The like-predicate tests whether a character value matches the pattern of another character value.
Expansion of Like-predicate

Syntax

Expansion of like-predicate

```
value-expression
   ────► LIKE 'pattern' ────────┐
   ┌─ NOT LIKE 'pattern' ─────┐
   │ host-variable ────┐
   │ special-register ───┐
   │ dynamic-parameter-marker ──┐
   │ routine-parameter ─────┘
   └─ ESCAPE 'escape-character' ────────────┐
       ┌─ host-variable ─┐
       │ dynamic-parameter-marker ──┐
       │ routine-parameter ─────┘
       └─ local-variable ──────┘
```

Parameters

value-expression

Specifies a value to be tested against a pattern or other value.

**Value-expression** must have a data type of CHARACTER, VARCHAR, BINARY, GRAPHIC, or VARGRAPHIC.

**NOT**

Reverses the test. **NOT** directs CA IDMS to test whether a specified value does not match the specified pattern.

**LIKE**

Identifies the pattern to which the value being tested is compared.

'pattern'

Specifies a character string literal to be used as the test pattern.

A test pattern can include wildcard characters:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ (underscore)</td>
<td>Represents any single character</td>
</tr>
<tr>
<td>% (percent sign)</td>
<td>Represents any string of zero or more characters</td>
</tr>
</tbody>
</table>

Wildcard characters can be used in any combination and any number of times in a test pattern.

G'graphics-pattern'

Specifies a double-byte character literal to be used as a test pattern.

Wildcard characters can be used as described for pattern except that the wildcard characters are the double-byte equivalents of the characters shown in the table above, and they are used to represent double-byte characters.
Expansion of Like-predicate

**host-variable**

Identifies a host variable that contains the character value to be used as the test pattern. The host variable must have been declared in an SQL declaration section and must be an elementary item instead of a group field.

**special-register**

Identifies a special register that contains the value to which the value being tested is compared.

CURRENT TIMEZONE may not be specified for special-register. For expanded special-register syntax, see Expansion of Special-register.

**dynamic-parameter-marker**

Indicates that a dynamic parameter is used to contain the character value for the test pattern.

**host-variable**

Identifies a local variable that contains the character value to be used as the test pattern. The local variable must have been declared in an SQL declaration statement.

**routine-parameter**

Identifies a routine parameter that contains the character value to be used as the test pattern. The routine parameter must have been defined in the parameter-definition of the SQL routine.

**local-variable**

Specifies a local variable to be used in the value-expression.

**ESCAPE**

The ESCAPE option allows the designation of an escape character for the pattern. The option must specify a one byte character value. If it appears in the pattern string, the escape character must be immediately followed by either a wildcard character or by another instance of the escape character. When this happens, the leading escape character is dropped from the match and the following character (wildcard or escape) is treated at face value instead of as a special character. For example, LIKE 'A_\%' matches all values beginning with A, while LIKE 'AZ_\%' ESCAPE 'Z' matches all values beginning with A_.

**Important!** Escape characters are not supported in installations with active DBCS support.

**'escape character'**

Specifies the character to be used as the escape character. Escape-character must be a one-byte character value.

**host-variable**

Specifies the character to be used as the escape character. Escape-character must be a one-byte character field.
**Dynamic-parameter-marker**

Specifies that the one-byte escape character is supplied through a dynamic parameter.

**Routine-parameter**

Specifies the character to be used as the escape character. *Escape-character* must be a one-byte character field.

**Local-variable**

Specifies the character to be used as the escape character. *Escape-character* must be a one-byte character field.

### Usage

**Truth Value of a LIKE Predicate without NOT**

The result of a LIKE predicate that does not include NOT is:

- True when the value being tested matches the test pattern
- False when the value being tested does not match the test pattern
- Unknown when either the value being tested or the test pattern is null

**Truth Value of a LIKE Predicate with NOT**

The result of a LIKE predicate that includes NOT is:

- True when the value being tested does not match the test pattern
- False when the value being tested matches the test pattern
- Unknown when either the value being tested or the test pattern is null

### Equivalence

If 'pattern' contains no wildcard character, the LIKE predicate is the equivalent of a comparison predicate using an equal sign, with the restriction that the lengths of the two values being compared must be identical. This restriction distinguishes a test for a match from a test for equality.
Expansion of Like-predicate

Evaluation of Trailing Blanks

If value-expression containing a character string with trailing blanks is compared to the same character string without trailing blanks in a LIKE predicate, the result is false. For example, 'ABC ' is not like 'ABC'. Similarly, 'ABC' is not like 'ABC'.

Graphics and Character Values

If value-expression is a character value, then the search pattern must also be a character value. If value-expression is a graphics value, then the search pattern must also be a graphics value.

Using Host Variables and Dynamic Parameters As Test Patterns

The value of the test pattern can be supplied through a host variable or a dynamic parameter. The value of the variable or parameter can include wildcard characters as described above. For example, assume you code the following:

```
02  PATTERN   PIC X(10).

....
MOVE ' %ABC%' TO PATTERN.
....
SELECT .... WHERE .... LIKE :PATTERN;
```

The pattern being used is '%ABC% ', which means 0 to n of anything followed by ABC, followed by 0 to n of anything, followed by 5 spaces. This doesn't yield the same result as:

```
SELECT .... WHERE .... LIKE : %ABC% ;
```

which means 0 to n of anything, followed by ABC, followed by 0 to n of anything.
Examples

Using the Underscore in a Pattern

The following SELECT statement identifies the consultants working on projects with four-character identifiers where the middle two characters are 2 and 0:

```sql
select con_id, con_lname
    from consultant
    where proj_id like '_20_';
```

Using the Percent Sign in a Pattern

The following SELECT statement identifies all employees whose last names begin with A or B:

```sql
select emp_fname, emp_lname, dept_id
    from employee
    where emp_lname like 'A%'
    or emp_lname like 'B%';
```

More Information

- For more information about character string literals, see Literals.
- For more information about host variables, see Host Variables.
- For more information about declaring local variables, see Local Variables.
- For more information about declaring routine-parameters, see Routine Parameters.

More information:

- Local Variables (see page 81)
- Host Variables (see page 77)
- Literals (see page 75)
- Routine Parameters (see page 84)

Expansion of Null-predicate

The null-predicate tests whether a value in a column is null.

Syntax

```
value-expression IS [NOT] NULL
```
Expansion of Quantified-predicate

Parameters

value-expression

    Specifies the value to be tested.

IS NULL

    Directs CA IDMS to test for the presence of a null value.

NOT

    Reverses the test. NOT directs CA IDMS to test for the presence of a non-null value.

Usage

Truth Value of a NULL Predicate without NOT

The result of a NULL predicate that does not include NOT is:

- True when the value being tested is null
- False when the value being tested is not null

Truth Value of a NULL Predicate with NOT

The result of a NULL predicate that includes NOT is:

- True when the value being tested is not null
- False when the value being tested is null

Example

As the Search Condition in a WHERE Parameter

The following SELECT statement identifies employees for whom no telephone number has been stored in the database:

```sql
select emp_id
    from employee
    where phone is null;
```

Note: For more information about search conditions, see Expansion of Search-condition, (see page 223)

Expansion of Quantified-predicate

The quantified-predicate tests the comparison of a value to either some or all the values in a specified set.
Expansion of Quantified-predicate

Chapter 6: Predicates and Search Condition

Syntax

Expansion of quantified-predicate

```
value-expression

comparison-operator ALL ( subquery ) SOME

ANY
```

Parameters

value-expression

Specifies a value to be compared to the set of values. For expanded value-expression syntax, see Expansion of Value-expression.

comparison-operator

Specifies the comparison operator to be used in the test. Valid values for comparison-operator are:

<table>
<thead>
<tr>
<th>Comparison operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Not less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than</td>
</tr>
<tr>
<td>ALL</td>
<td>Directs CA IDMS to test whether the specified value relates to all the values in the test set in the way specified by the comparison operator.</td>
</tr>
<tr>
<td>SOME/ANY</td>
<td>Directs CA IDMS to test whether the specified value relates to at least one value in the test set in the way specified by the comparison operator. SOME and ANY are synonyms.</td>
</tr>
</tbody>
</table>

( subquery )

Specifies a subquery that returns zero or more rows and whose result table consists of a single column. For expanded subquery syntax, see Expansion of Subquery.
Usage

Comparable Data Types

The data types of the values being compared must be comparable.

Note: For more information about comparing values of different data types, see Comparison, Assignment, Arithmetic, and Concatenation Operations.

Truth Table for a Quantified Predicate with ALL

The result of a quantified predicate that includes ALL is:

- True when the value being tested relates to every value in the test set in the way specified by the comparison operator or when the result of the subquery is an empty set
- False when the value being tested does not relate to at least one value in the test set in the way specified by the comparison operator
- Unknown when the value being tested is null or when at least one value in the test set is null and the value being tested relates to all other values in the test set in the way specified by the comparison operator

Truth Table for a Quantified Predicate with SOME or ANY

The result of a quantified predicate that includes SOME or ANY is:

- True when the value being tested relates to at least one value in the test set in the way specified by the comparison operator
- False when the value being tested does not relate to any value in the test set in the way specified by the comparison operator or when the result of the subquery is an empty set
- Unknown when the value being tested is null or when at least one value in the test set is null and the value being tested does not relate to any value in the test set in the way specified by the comparison operator
Examples

Using ALL

The following SELECT statement identifies the employees whose percent of salary increase at their 1999 review was greater than their percent of salary increase in any other year:

```
select emp_id
  from benefits b1
  where fiscal_year = '99'
    and review_percent > all
      (select review_percent
        from benefits b2
        where b1.emp_id = b2.emp_id
        and fiscal_year <> '99');
```

Using ANY

The following SELECT statement identifies employees who earned more in commission in the 1999 fiscal year than they did in salary in at least one fiscal year:

```
select s.emp_id
  from sales s, position p1
  where s.emp_id = p1.emp_id
    and s.fiscal_year = '99'
    and comm_percent * sales_to_date > any
      (select salary_amount
        from position p2
        where s.emp_id = p2.emp_id);
```

Expansion of Search-condition

The search-condition represents a truth value in an SQL statement.
Expansion of Search-condition

Syntax

Expansion of search-condition

```
NOT
  between-predicate
  comparison-predicate
  exists-predicate
  in-predicate
  like-predicate
  null-predicate
  quantified-predicate
  ( search-condition )

AND
  NOT
  OR

between-predicate
  comparison-predicate
  exists-predicate
  in-predicate
  like-predicate
  null-predicate
  quantified-predicate
  ( search-condition )
```

Parameters

**NOT**

Reverses the truth value, if known, of the operand that follows; that is:

- A true value becomes false
- A false value becomes true
- An unknown value remains unknown

**between-predicate**

Represents the truth value resulting from the evaluation of a BETWEEN predicate. For expanded between-predicate syntax, see Expansion of Between-predicate.

**comparison-predicate**

Represents the truth value resulting from the evaluation of a comparison predicate. For expanded comparison-predicate syntax, see Expansion of Comparison-predicate.

**exists-predicate**

Represents the truth value resulting from the evaluation of an EXISTS predicate. For expanded exists-predicate syntax, see Expansion of Exists-predicate.

**in-predicate**

Represents the truth value resulting from the evaluation of an IN predicate. For expanded in-predicate syntax, see Expansion of In-predicate.

**like-predicate**

Represents the truth value resulting from the evaluation of a LIKE predicate. For expanded like-predicate syntax, see Expansion of Like-predicate.
null-predicate

Represents the truth value resulting from the evaluation of a NULL predicate. For expanded null-predicate syntax, see Expansion of Null-predicate.

quantified-predicate

Represents the truth value resulting from the evaluation of a quantified predicate. For expanded quantified-predicate syntax, see Expansion of Quantified-predicate.

(search-condition)

Specifies another search condition to be used as a single operand in the search condition. To be manipulated as a single operand, the search condition must be enclosed in parentheses.

AND

Specifies that both the operand preceding the operator and the operand following the operator must be true for the search condition to be true.

OR

Specifies that either the operand preceding the operator, the operand following the operator, or both operands must be true for the search condition to be true.

Usage

A search condition in an SQL statement specifies criteria used to restrict the data processed by the statement:

- In a WHERE parameter, the search condition restricts the rows processed by the statement.
  The WHERE parameter occurs in query-specification and in the DELETE, SELECT, and UPDATE statements.
- In a HAVING parameter, the search condition restricts the table groupings processed by the statement.
  The HAVING parameter occurs in query-specification and the SELECT statement.
- In a CHECK or ADD CHECK parameter, the search condition restricts the data that can be stored in a table. A search condition in a CHECK or ADD CHECK parameter is also called a check constraint.
  The CHECK parameter occurs in the CREATE TABLE statement. The ADD CHECK parameter occurs in the ALTER TABLE statement.
Restrictions on search-condition in a CHECK or ADD CHECK Parameter

In the CHECK parameter of the CREATE TABLE statement or the ADD CHECK parameter of the ALTER TABLE statement:

- The search condition cannot include any host variables, routine parameters, local variables, user-defined-functions, aggregate functions, EXISTS predicates, quantified predicates, subqueries, or dynamic parameter markers.
- Column references in the search condition must identify columns in the table being created or altered.

Truth Values

The result of a search condition is one of three possible truth values: true, false, or unknown. The unknown value occurs only when the search condition includes one or more null values.

CA IDMS obtains the result by evaluating the search condition for a particular row in a table or a particular table grouping. Processing occurs according to the results, as described in the following table:

<table>
<thead>
<tr>
<th>If the result is:</th>
<th>CA IDMS:</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>Continues processing the statement for the row or group</td>
</tr>
<tr>
<td>False</td>
<td>CA IDMS does not process the statement for the row or group</td>
</tr>
<tr>
<td>Unknown</td>
<td>Does not process the row when the search condition occurs in the WHERE or HAVING parameters of a SELECT, UPDATE, or DELETE statement, or query specification</td>
</tr>
<tr>
<td></td>
<td>Processes the row when the search condition occurs in a CHECK clause that is tested during an INSERT or UPDATE operation</td>
</tr>
</tbody>
</table>

Truth Table for AND

The result of the AND operation for each possible combination of operands is given by the following truth table:

<table>
<thead>
<tr>
<th>AND</th>
<th>True</th>
<th>False</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
<td>Unknown</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>
Expansion of Search-condition

Chapter 6: Predicates and Search Condition

227

Truth Table for OR

The result of the OR operation for each possible combination of operands is given by the following truth table:

<table>
<thead>
<tr>
<th>OR</th>
<th>True</th>
<th>False</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>Unknown</td>
</tr>
<tr>
<td>Unknown</td>
<td>True</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

Order of Evaluation

After evaluating the individual operands, CA IDMS performs the operations in a search condition in the following order:

1. The unary operation NOT from left to right
2. AND from left to right
3. OR from left to right

You can use parentheses to override the default order of evaluation. Operations in parentheses are performed first.

For example, assuming the value in :SALARY is 35,000, the result of the following search condition is true:

:salary > 20000 or :salary = 0 and :salary < 30,000

When the OR operation is enclosed in parentheses, the result of the expression is false:

(:salary > 20000 or :salary = 0) and :salary < 30,000
Examples

A Single Operand

The following ALTER TABLE statement directs CA IDMS to store only values less than or equal to 10 in the BONUS_PERCENT column of the POSITION table. The search condition in the ADD CHECK parameter consists of a single operand (a comparison predicate).

```
alter table position
  add check (bonus_percent <= 10);
```

Two Operands with OR

The following SELECT statement returns the number of employees in each department that has either five or more employees or no employees:

```
select dept_id, count(emp_id)
  from employee
  group by dept_id
  having count(emp_id) >= 5
    or count(emp_id) = 0;
```

Multiple Operands

The following SELECT statement identifies the project leaders of projects that were scheduled to have started by now but have not and that have no assigned employees. The search condition in the first WHERE parameter includes three operands. The first is a comparison predicate, the second is a NULL predicate, and the third is an EXISTS predicate with the unary operator NOT.

```
select proj_leader_id
  from project p
    where est_start_date < current date
      and act_start_date is null
      and not exists
        (select emp_id
          from employee e
            where e.proj_id = p.proj_id);
```
More Information

- For more information about the WHERE parameter, see:
  - Expansion of Subquery
  - Expansion of Query-specification
  - DELETE, SELECT, UPDATE

- For more information about the HAVING parameter, see:
  - Expansion of Subquery
  - Expansion of Query-specification
  - SELECT or UPDATE

- For more information about the CHECK parameter, see CREATE TABLE.

- For more information about the ADD CHECK parameter, see ALTER TABLE.
Chapter 7: Query Specifications, Subqueries, Query Expressions, and Cursor Specifications

This section contains the following topics:

- **Query Specifications** (see page 231)
- **Expansion of Query-specification** (see page 231)
- **Subqueries** (see page 238)
- **Expansion of Subquery** (see page 239)
- **Query Expressions** (see page 240)
- **Expansion of Query-expression** (see page 241)
- **Expansion of Cursor-specification** (see page 244)

**Query Specifications**

A query specification is the form of the SELECT statement used to represent a table of values in an SQL statement. The values represented by a query specification are derived from the tables, views, procedures and table procedures named in the FROM parameter of the query specification.

**Expansion of Query-specification**

The expansion of query-specification represents a table to be used in the evaluation of an SQL statement.

**Syntax**

```
Expansion of query-specification

SELECT ALL | DISTINCT
*
value-expression .
result-name

AS

table-name.*
alias.*

FROM

AS

table-reference
(query-expression)

AS

alias
```
Parameters

**ALL**
Directs CA IDMS to return all the rows, including duplicates, in the requested result table. ALL is the default when you specify neither ALL nor DISTINCT.

**DISTINCT**
Directs CA IDMS to eliminate duplicate rows from the result table of the query specification.

** Specifies that the result table is to include all columns in the tables, views, procedures and table procedures named in the FROM parameter of the query specification. The columns in the tables, views, procedures and table procedures are concatenated in the order in which the tables and views are specified in the FROM parameter.

**value-expression**
Identifies the values to be included in a result column. Typically, value-expression includes a column reference.

Each column reference in value-expression must identify a column in a table named in the FROM parameter of the query specification.

The number of columns in a result table is the same as the number of value expressions in the query specification defining the result table. For expanded value-expression syntax, see Expansion of Value-expression.

**AS result-name**
Specifies a name for the result column identified by value-expression. Result-name must be a 1- through 32-character name that follows the conventions for SQL identifiers.
table-name.*

Specifies that the result table is to include all columns in the table identified by table-name.

Table-name must match an occurrence of table-name in the FROM parameter.

alias.*

Specifies that the result table is to include all columns in the table identified by alias.

Alias must match an occurrence of alias in the FROM parameter.

FROM table-reference

Identifies one or more tables, views, procedures or table procedures from which the result table is to be derived. For expanded table-reference syntax, see Expansion of Table-reference.

(query-expression)

Represents a table to be used in the evaluation of an SQL statement.

AS alias

Defines a new name to be used to identify the table, view, procedure or table procedure within the query specification. Alias must be a 1- through 18-character name that follows the conventions for SQL identifiers.

WHERE

Introduces criteria that a row must meet to be included in the result table.

search-condition

Specifies the set of values against which a row is tested:

- When the value of search-condition is true, the row is included in the result table
- When the value of search-condition is false or unknown, the row is not included in the result table

For expanded search-condition syntax, see Expansion of Search-condition.

extended-search-condition

Specifies a search condition that includes a set specification. For expanded extended-search-condition syntax, see Expansion of Extended-search Condition.
PRESERVE

Requests an outer join on the specified table, view, or table procedure. The PRESERVE parameter is a CA IDMS extension of the SQL standard.

To specify a more powerful outer join that is compatible with the SQL standard, use the joined-table construct as table-reference.

table-name

Specifies by name the table, view, procedure or table procedure to be preserved in an outer join. For expanded table-name syntax, see Expansion of Table-name.

alias

Specifies the table, view, procedure or table procedure to be preserved in an outer join by the alias defined for the table, view, procedure or table procedure in the FROM parameter of the query specification.

GROUP BY column-name

Groups the rows in the table defined by the FROM and WHERE parameters by the values in the specified columns. Rows with the same value in each grouping column are grouped together.

Column-name must identify a column in a table, view, procedure or table procedure named in the FROM parameter of the query specification. Multiple column names must be separated by commas.

table-name

Specifies the table, view, procedure or table procedure includes the named column. For expanded table-name syntax, see Expansion of Table-name.

alias

Specifies the alias associated with the table, view, procedure or table procedure that includes the named column. Alias must be defined in the FROM parameter of the query specification.

rowid-pseudo-column

Specifies a pseudo-column ROWID to be used as a grouping column. See Expansion of rowid-pseudo-column for more information.
HAVING search-condition

Specifies criteria a group must meet to be included in the result table:

- When the value of search-condition is true, the group is included in the result table
- When the value of search-condition is false or unknown, the group is not included in the result table

For expanded search-condition syntax, see Expansion of Search-condition.

OPTIMIZE FOR literal ROWS

Specifies the expected number of output rows from this query-specification. It is used by the optimizer to generate the best possible access strategy for satisfying query-expression. The string literal is an integer constant. The OPTIMIZE FOR parameter is a CA IDMS extension of the SQL standard.

Usage

Outer Join Using PRESERVE

Within query-specification, PRESERVE can be used to request an outer join on one of the tables, views, procedures or table procedures named in the FROM parameter. If PRESERVE is specified, the result table includes rows of the preserved table for which no matching row exists in the other tables used in the join operation.

If no matching row exists, the corresponding columns in the result table are set to null. Predicates in the WHERE clause other than those used to perform the outer join are evaluated before determining whether a matching row exists.

The following statement returns the names of all active employees. The name of the employee's spouse is also returned if found. The logic of the statement is that the result table includes the name of each active employee, and whether the employee has a spouse:

```
select e.first_name, e.last_name,
    s.first_name, s.last_name
from employee e, relation s
where e.empid=s.empid
    and e.status='A' -- active employee
    and s.relationship='S' -- employee's spouse
preserve e ;
```
Note: Outer join and many other join types can be specified to be compatible with the SQL standard using the joined-table construct in table-reference. See Expansion of Table-reference, for more information.

PRESERVE and Column Order

When using PRESERVE and specifying "**" as the result column list, the order of the columns in the result table depend on which table is being preserved. The columns of the preserved table are always first.

Value Expressions without Column References

If the value expression that identifies a result column does not include any column references, the result column contains the same value in each row. This value is derived directly from the value expression without reference to the table defined by the FROM parameter of the query specification.

Uniqueness of Table References

Each alias and each table reference without an associated alias must be unique within the FROM parameter of a query specification.

Column References in the WHERE Parameter

Each column reference directly included in the search condition in the WHERE parameter of a query specification must unambiguously identify a column in a table, view, procedure or table procedure specified in the FROM parameter of the query specification, or must be an outer reference.

Note: For information about outer references, see Subqueries.

GROUP BY Parameter Requirements

When a query specification includes the GROUP BY parameter, each column reference in the value expressions that identify the result columns must either identify a column specified in the GROUP BY parameter or occur only in the argument of an aggregate function. If the result columns are identified by an asterisk (*), the GROUP BY parameter must include all the columns in the tables, views, and table procedures specified in the FROM parameter.

Query Specifications without the GROUP BY Parameter

If a query specification does not include the GROUP BY parameter and any column reference in a value expression that identifies a result column is included in the argument of an aggregate function:

- All column references in all the value expressions must be in aggregate functions
- The entire table defined by the FROM and WHERE parameters is treated as a single group
Column References in the HAVING Parameter

Each column reference included in the search condition in the HAVING parameter of a query specification must either identify a column specified in the GROUP BY parameter of the query specification, occur in the argument of an aggregate function, or be an outer reference.

When to Use OPTIMIZE FOR Literal ROWS

Under some circumstances, the SQL optimizer may choose a less than optimal access strategy to satisfy a query expression. This typically happens with host program embedded SQL statements which contain WHERE clauses with host variable references, rather than explicit constants. For example, a BETWEEN clause involving host variables may induce the optimizer to assume many rows will be retrieved, causing it to choose an area sweep to satisfy the request. Without knowing the underlying values of the host variables, the optimizer cannot know if the BETWEEN will always qualify a small number of rows, thus possibly making an index retrieval much more efficient. The OPTIMIZE FOR literal ROWS clause is used to override the number of expected rows deduced by the optimizer. This allows it to generate better access strategies.

Examples

In a CREATE VIEW Statement

The following CREATE VIEW statement defines a view derived from three tables:

```sql
create view former_employee
as select e.emp_id, emp_fname, emp_lname, job_title, start_date, finish_date
from employee e, job j, position p
where e.emp_id = j.emp_id
    and e.emp_id = p.emp_id
    and finish_date is not null;
```

In an INSERT Statement

The following INSERT statement inserts rows into the TEMP_EMP_SKILL table.

```sql
insert into temp_emp_skill
    select emp.emp_id, dept_id, skill_name, skill_level
from employee emp, expertise exp, skill s
where emp.emp_id = exp.emp_id
    and exp.skill_id = s.skill_id;
```
Subqueries

A subquery is a query specification used in predicates or in the SET clause of the UPDATE statement or in XML-value-expression. Each subquery in a predicate represents a set of zero or more values to be used in the test specified by the predicate. A subquery used in the SET clause of the UPDATE statement or in XML-value-expression represents either the NULL value or a single value. The values represented by a subquery are derived from the query specification of the subquery.

Subqueries are always enclosed in parentheses, except when used as XML-value-expression.

Nesting Subqueries

You can nest subqueries in an SQL statement. For example, a subquery in the WHERE parameter of a SELECT statement can include another subquery in its own WHERE or HAVING parameter:

```sql
select ... where ... (select ... where ... (select ...) );
```

Outer References

An outer reference is a reference to a column named in an outer subquery, an outer query specification, or the SELECT statement where the subquery occurs.

For example, with reference to the illustration above:

- If Subquery 2 contains a reference to a column named in Subquery 1, it is an outer reference
- If Subquery 2 contains a reference to a column named in the SELECT statement, it is an outer reference
- If Subquery 1 contains a reference to a column named in the SELECT statement, it is an outer reference
Correlated Subqueries

A correlated subquery is a subquery that contains an outer reference.

CA IDMS must evaluate a correlated subquery once for each value in the outer-reference column. The result of the evaluation differs depending on the value in the outer-reference column.

In contrast, CA IDMS must evaluate a subquery that does not include any outer references only once.

For an example of a correlated subquery, see A Correlated Subquery in a Comparison Predicate.

Expansion of Subquery

The expansion of subquery specifies a set of values to be used in the evaluation of a predicate or the SET clause of an UPDATE statement.

Syntax

```
Expansion of subquery
  query-specification
```

Parameters

query-specification

Specifies the query specification that comprises the subquery. For expanded query-specification syntax, see Expansion of Query-specification.

Usage

Restriction on DISTINCT

You can specify DISTINCT only once in a subquery (not counting occurrences in nested subqueries). For example, if the value expression that identifies the result column includes an aggregate function with the keyword DISTINCT, you cannot specify DISTINCT either before the value expression or with any other aggregate function.

Column References in the WHERE parameter

Each column that the query specification of a subquery references must identify a column of a table, view, procedure or table procedure named in the FROM clause of the query specification or be an outer reference.
Examples

A Subquery Without Correlation in an IN Predicate

The following SELECT statement returns the name and department identifier of each employee who has more than 80 hours of outstanding vacation time. The set of values returned by the subquery consists of the identifiers of all employees with more than 80 hours of outstanding vacation time.

\[
\text{select emp\_fname, emp\_lname, dept\_id}
\text{from employee}
\text{where emp\_id in}
\text{(select emp\_id}
\text{from benefits}
\text{group by emp\_id}
\text{having sum(vac\_accrued) - sum(vac\_taken) > 80);}
\]

A Correlated Subquery in a Comparison Predicate

The following SELECT statement identifies employees who earn more than their managers. The subquery is evaluated once for each value in the EMP_ID column of the EMPLOYEE table named in the outer SELECT statement.

\[
\text{select e1.emp\_id}
\text{from employee e1, position p1}
\text{where e1.emp\_id = p1.emp\_id}
\text{and p1.salary\_amount >}
\text{(select p2.salary\_amount}
\text{from employee e2, position p2}
\text{where e1.manager\_id = e2.emp\_id}
\text{and e2.emp\_id = p2.emp\_id);}
\]

More Information

- For more information about aggregate functions in subqueries, see Aggregate-function.
- For more information about subqueries in comparison predicates, see Expansion of Comparison-predicate.

Query Expressions

A query expression is an expression used to represent a table of values in an SQL statement. The operands in a query expression are tables represented by query specifications. A query expression can include one or more query specifications. Each query specification is linked to the next by the UNION operator. The data lengths of the unioned columns must be identical.
Result of a Query Expression

The table resulting from the evaluation of a query expression is derived from the concatenation of the rows in the tables defined by the operands.

Note: For more information about other types of expressions, see Values and Value Expressions.

Expansion of Query-expression

The expanded parameters of query-expression represent a table to be used in the evaluation of an SQL statement.

Syntax

```
query-specification
( query-expression )
```

Parameters

**query-specification**

Represents a table resulting from the evaluation of a query specification. For expanded **query-specification** syntax, see Expansion of Query-specification.

**( query-expression )**

Specifies another query expression to be used as a single operand in the query expression.

**UNION all**

Specifies that:

- The result table is to include the rows from the table represented by the operand preceding the UNION operator and the rows from the table represented by the operand following the UNION operator.
- Duplicate rows are eliminated from the table resulting from the UNION operation, unless the ALL keyword is present.

The data types and lengths of unioned columns must be compatible. Detailed information is presented under "Usage".

**ALL**

Specifies that all rows from the UNION operation are retained; duplicates are not discarded.
### Usage

#### Result Data Type

This matrix shows the data type that results when a UNION operation is performed on columns of compatible data types.

<table>
<thead>
<tr>
<th></th>
<th>I2</th>
<th>I4</th>
<th>I8</th>
<th>R4</th>
<th>R8</th>
<th>PD</th>
<th>ZD</th>
<th>UP</th>
<th>UZ</th>
<th>CH</th>
<th>BI</th>
<th>DT</th>
<th>GR</th>
<th>VG</th>
<th>TI</th>
<th>DI</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>I2</td>
<td>I4</td>
<td>I8</td>
<td>R4</td>
<td>R8</td>
<td>PD</td>
<td>ZD</td>
<td>PD</td>
<td>ZD</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TI</td>
<td>DI</td>
<td></td>
</tr>
<tr>
<td>I4</td>
<td>I4</td>
<td>I4</td>
<td>I8</td>
<td>R4</td>
<td>R8</td>
<td>PD</td>
<td>ZD</td>
<td>PD</td>
<td>ZD</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TI</td>
<td>DI</td>
<td></td>
</tr>
<tr>
<td>I8</td>
<td>I8</td>
<td>I8</td>
<td>I8</td>
<td>R4</td>
<td>R8</td>
<td>PD</td>
<td>ZD</td>
<td>PD</td>
<td>ZD</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TI</td>
<td>DI</td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TI</td>
<td>DI</td>
<td></td>
</tr>
<tr>
<td>R8</td>
<td>R8</td>
<td>R8</td>
<td>R8</td>
<td>R8</td>
<td>R8</td>
<td>R8</td>
<td>R8</td>
<td>R8</td>
<td>R8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TI</td>
<td>DI</td>
<td></td>
</tr>
<tr>
<td>PD</td>
<td>PD</td>
<td>PD</td>
<td>PD</td>
<td>R4</td>
<td>R8</td>
<td>PD</td>
<td>ZD</td>
<td>PD</td>
<td>ZD</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TI</td>
<td>DI</td>
<td></td>
</tr>
<tr>
<td>ZD</td>
<td>ZD</td>
<td>ZD</td>
<td>ZD</td>
<td>R4</td>
<td>R8</td>
<td>PD</td>
<td>ZD</td>
<td>PD</td>
<td>ZD</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TI</td>
<td>DI</td>
<td></td>
</tr>
<tr>
<td>UP</td>
<td>PD</td>
<td>PD</td>
<td>PD</td>
<td>R4</td>
<td>R8</td>
<td>PD</td>
<td>ZD</td>
<td>UP</td>
<td>UZ</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TI</td>
<td>DI</td>
<td></td>
</tr>
<tr>
<td>UZ</td>
<td>ZD</td>
<td>ZD</td>
<td>ZD</td>
<td>R8</td>
<td>R8</td>
<td>PD</td>
<td>ZD</td>
<td>UP</td>
<td>UZ</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TI</td>
<td>DI</td>
<td></td>
</tr>
<tr>
<td>CH</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CH</td>
<td>VC</td>
<td>CH</td>
<td>GR</td>
<td>VG</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VC</td>
<td>VC</td>
<td>VC</td>
<td>VG</td>
<td>VG</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>BI</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DT</td>
<td></td>
</tr>
<tr>
<td>DT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>GR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CH</td>
<td>VC</td>
<td>BI</td>
<td>GR</td>
<td>VG</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>VG</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VC</td>
<td>VC</td>
<td>BI</td>
<td>VG</td>
<td>VG</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>TI</td>
<td>TI</td>
<td>TI</td>
<td>TI</td>
<td>TI</td>
<td>TI</td>
<td>TI</td>
<td>TI</td>
<td>TI</td>
<td>TI</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TI</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>DI</td>
<td>DI</td>
<td>DI</td>
<td>DI</td>
<td>DI</td>
<td>DI</td>
<td>DI</td>
<td>DI</td>
<td>DI</td>
<td>DI</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DI</td>
<td></td>
</tr>
</tbody>
</table>

**Key:**
- I2—Small integer
- I4—Integer
- I8—Long integer
- R4—Real
- R8—Double precision
- PD—Decimal
- ZD—Numeric
- CH—Character
- VC—Varchar
- BI—Binary
- DT—Date/time
- UP—Unsigned decimal
- UZ—Unsigned numeric
- GR—Graphic
- VG—Vargraphic
- TI—Time interval
- DI—Date interval
- --Incompatible types
Nullable Columns

If both columns in a UNION operation are not nullable, the result is not nullable; otherwise the result is nullable.

Result Precision

The result precision of decimal, numeric, char, varchar, graphic, vargraphic, and binary is always large enough to hold the larger of the source columns in a UNION operation.

Restrictions on Multiple Query Specifications

If a query expression includes more than one query specification:

- Result tables returned by the query specifications must all have the same number of columns
- Columns in any given position in the result tables returned by the query specifications must be compatible for assignment

Updateable Query Expressions

A query expression is updateable under the following conditions:

- The expression consists of a single query specification (that is, the query expression does not include the UNION operator)
- The FROM parameter in the query specification specifies only one table, view, procedure or table procedure
- If a view is named in the FROM parameter, it is updateable
- The query specification does not contain DISTINCT, PRESERVE, GROUP BY, or HAVING parameters, nor is an aggregate function used in the specification of a result column

Note: For more information about usage considerations for query expressions, see "Usage" under Expansion of Query-specification
**Example**

In a DECLARE CURSOR Statement

The following DECLARE CURSOR statement creates a cursor for the table resulting from the UNION of two query specifications. The four result columns identified by the second query specification have the same data types, lengths, and null specifications as the four result columns identified by the first query specification.

```sql
declare all_curr_emp cursor
  for select emp_id, emp.fname, emp.lname, dept_id
    from employee
    where status <> 'T'
  union select con_id, con.fname, con.lname, dept_id
    from consultant
    where proj_id is not null
  order by 4, 1, 2, 3;
```

**Expansion of Cursor-specification**

The expanded parameters of cursor-specification represent the body of a cursor definition.

**Syntax**

```sql
Expansion of cursor-specification

query-expression order-by-specification

Expansion of order-by-specification

ORDER BY table-name alias.
  column-number result-name rowid-pseudo-column
```

```sql
FOR READ ONLY UPDATE OF column-name
```
Parameters

Expansion of cursor-specification

query-expression

Represents a table resulting from the evaluation of a query-expression. For expanded syntax, see Expansion of Query-expression.

order-by-specification

Specifies a sort order for the rows in the result table defined by query-expression. Expanded syntax for order-by-specification is shown above, immediately following the cursor-specification syntax.

FOR READ ONLY

Specifies the cursor associated with this cursor-expression is used for retrieval operations only. If specified, it prohibits the execution of both positioned UPDATEs and DELETEs that reference the cursor.

FOR UPDATE

Specifies that the cursor is used for positioned UPDATE operations.

OF column-name

Identifies a column that may be updated through positioned UPDATE statements. If no columns are specified, then all columns in the table may be updated.

Expansion of order-by-specification

ORDER BY

Sorts the rows in the result table defined by query-expression in ascending or descending order by the values in the specified columns. Rows are ordered first by the first column specified, then by the second column specified with the ordering established by the first column, then by the third column specified, and so on.

column-name

Specifies a sort column by name. Column-name must identify a column in the result table of the query expression.

table-name

Specifies the table, view, procedure or table procedure that includes the named column. For Expansion of Table-name expanded table-name syntax, see Identifying Entities in Schemas (see page 42).

alias

Specifies the alias associated with the table, view, procedure or table procedure that includes the named column. Alias must be defined in the FROM parameter of the query specification that makes up the query expression.
Expansion of Cursor-specification

**column-number**

Specifies a sort column by the position of the column in the result table. The first result column is in position 1.

*Column-number* must be an integer in the range 1 through the number of columns in the result table.

**result-name**

Specifies the sort column by the result name specified in the AS parameter of the query expression.

**rowid-pseudo-column**

Specifies the sort column as a ROWID pseudo-column. For expanded syntax, see Expansion of rowid-pseudo-column.

**ASC**

Indicates that the values in the specified column are to be sorted in ascending order. ASC is the default.

**DESC**

Indicates that the values in the specified column are to be sorted in descending order.

**Usage**

**Updateable cursors**

A cursor defined by a cursor specification is updateable if the cursor specification:

- Contains an updateable query-expression
- Does not contain an ORDER BY clause
- Does not contain a FOR READ ONLY clause

Updateable cursors may be referenced in positioned DELETE statements.

To reference a cursor in a positioned UPDATE statement, it must be updateable and the FOR UPDATE clause must be specified within the cursor specification.

**Note:** To ensure optimal performance when processing a cursor that is referenced in a positioned UPDATE statement, you should explicitly identify the columns to be updated rather than specifying FOR UPDATE without naming the columns.
Example

Defining a Cursor for retrieval-only

The following DECLARE CURSOR statement defines a static cursor by including a cursor-specification directly. In this case, the cursor being defined can only be used to retrieve rows from the database. By coding the FOR READ ONLY option, you ensure that neither positioned UPDATEs nor DELETEs are allowed against the cursor:

```sql
EXEC SQL
  DECLARE EMP_CURSOR CURSOR FOR
    SELECT EMP_ID, DEPT_ID, EMP_LNAME
    FROM EMPLOYEE
    FOR READ ONLY
END-EXEC
```

Defining a Dynamic Cursor for UPDATE Operations

The following set of code defines a dynamic cursor to be used to update the DEPT_ID column of the EMPLOYEE table. The cursor-specification containing the FOR UPDATE clause is first prepared and then an ALLOCATE CURSOR statement is used to create the cursor:

```sql
MOVE 'SELECT *
    FROM EMPLOYEE FOR UPDATE OF DEPT_ID'
  TO ST-TEXT.

EXEC SQL
  PREPARE 'EMP-STATEMENT' FROM :ST-TEXT
END-EXEC

EXEC SQL
  ALLOCATE 'EMP-CURSOR' CURSOR FOR 'EMP-STATEMENT'
END-EXEC
```
This section contains the following topics:

- **Statement Categories** (see page 251)
- **ALLOCATE CURSOR** (see page 258)
- **ALTER ACCESS MODULE** (see page 261)
- **ALTER CATALOG** (see page 269)
- **ALTER CONSTRAINT** (see page 270)
- **ALTER FUNCTION** (see page 272)
- **ALTER INDEX** (see page 277)
- **ALTER PROCEDURE** (see page 279)
- **ALTER SCHEMA** (see page 285)
- **ALTER TABLE** (see page 290)
- **ALTER TABLE PROCEDURE** (see page 300)
- **BEGIN DECLARE SECTION** (see page 305)
- **CALL** (see page 306)
- **CLOSE** (see page 313)
- **COMMIT** (see page 315)
- **CONNECT** (see page 317)
- **CREATE ACCESS MODULE** (see page 320)
- **CREATE CALC** (see page 333)
- **CREATE CONSTRAINT** (see page 334)
- **CREATE FUNCTION** (see page 341)
- **CREATE INDEX** (see page 352)
- **CREATE KEY** (see page 357)
- **CREATE PROCEDURE** (see page 361)
- **CREATE SCHEMA** (see page 373)
- **CREATE TABLE** (see page 378)
- **CREATE TABLE PROCEDURE** (see page 387)
- **CREATE TEMPORARY TABLE** (see page 392)
- **CREATE VIEW** (see page 394)
- **DEALLOCATE PREPARE** (see page 401)
- **DECLARE CURSOR** (see page 402)
- **DECLARE EXTERNAL CURSOR** (see page 406)
- **DELETE** (see page 408)
- **DESCRIBE** (see page 413)
- **DESCRIBE CURSOR** (see page 416)
- **DROP ACCESS MODULE** (see page 418)
- **DROP CALC** (see page 420)
- **DROP CONSTRAINT** (see page 421)
- **DROP FUNCTION** (see page 423)
- **DROP INDEX** (see page 424)
- **DROP KEY** (see page 426)
- **DROP PROCEDURE** (see page 427)
- **DROP SCHEMA** (see page 428)
- **DROP TABLE** (see page 430)
- **DROP TABLE PROCEDURE** (see page 432)
- **DROP VIEW** (see page 433)
- **END DECLARE SECTION** (see page 434)
- **EXECUTE** (see page 435)
CA IDMS SQL statements fall into the following categories:

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access module management</td>
<td>Control the creation and characteristics of access modules</td>
</tr>
<tr>
<td>Authorization</td>
<td>Control access to and ownership of database entities</td>
</tr>
<tr>
<td>Control</td>
<td>Define the flow of control in an SQL routine and assign values from expressions to routine parameters or local variables</td>
</tr>
<tr>
<td>Data description</td>
<td>Control the creation and characteristics of logical database entities</td>
</tr>
</tbody>
</table>
Statement Categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data manipulation</td>
<td>Retrieve and update data in the database</td>
</tr>
<tr>
<td>Diagnostics &amp; Statistics</td>
<td>Diagnose the execution of SQL statements and return statistical information of the current transaction.</td>
</tr>
<tr>
<td>Dynamic compilation</td>
<td>Control the run-time compilation and execution of SQL statements</td>
</tr>
<tr>
<td>Precompiler directives</td>
<td>Instruct the precompiler to include specified data structures and to generate specified error-processing code</td>
</tr>
<tr>
<td>Session management</td>
<td>Establish and control the characteristics of SQL sessions</td>
</tr>
<tr>
<td>Transaction management</td>
<td>Establish and control the characteristics of CA IDMS database transactions</td>
</tr>
</tbody>
</table>

Access Module Management Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER ACCESS MODULE</td>
<td>Modifies an access module in the dictionary</td>
</tr>
<tr>
<td>CREATE ACCESS MODULE</td>
<td>Creates an access module from one or more SQL statement modules (RCMs)</td>
</tr>
<tr>
<td>DROP ACCESS MODULE</td>
<td>Deletes an access module and its definition from the dictionary</td>
</tr>
<tr>
<td>EXPLAIN</td>
<td>Describes the strategy used to access data for a DELETE, INSERT, SELECT, or UPDATE statement</td>
</tr>
</tbody>
</table>

Authorization Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT definition privileges</td>
<td>Gives one or more users the privilege of performing selected actions on a specified schema, access module, table, or view</td>
</tr>
<tr>
<td>GRANT execution privilege</td>
<td>Gives one or more users the privilege of executing a specified access module</td>
</tr>
<tr>
<td>GRANT all privileges</td>
<td>Gives one or more users all definition and access privileges on a specified table or view</td>
</tr>
</tbody>
</table>
### Statement Categories

#### Chapter 8: Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT table access privileges</td>
<td>Gives one or more users the privilege of performing selected actions on a specified table or view</td>
</tr>
<tr>
<td>REVOKE definition privileges</td>
<td>Removes from one or more users the privilege of performing selected actions on a specified schema, access module, table, or view</td>
</tr>
<tr>
<td>REVOKE execution privilege</td>
<td>Removes from one or more users the privilege of executing a specified access module</td>
</tr>
<tr>
<td>REVOKE all privileges</td>
<td>Removes from one or more users all definition and access privileges on a specified table or view</td>
</tr>
<tr>
<td>REVOKE table access privileges</td>
<td>Removes from one or more users the privilege of performing selected actions on a specified table or view</td>
</tr>
<tr>
<td>TRANSFER OWNERSHIP</td>
<td>Passes ownership of a schema from one user or group of users to another</td>
</tr>
</tbody>
</table>

### Control Statements

The CA IDMS SQL Control statements allow you to define the flow of control in an SQL routine and assign values to routine parameters or local variables. These statements are used primarily by the following:

- Developers of SQL routines
- Programmers developing SQL application programs
- Users of interactive SQL tools

**Note:** For more information about the individual Control statements, see Control Statements.

### Data Description Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER CATALOG</td>
<td>Supports the correct sorting of additional national characters for specific languages (used by the alternate character set feature)</td>
</tr>
<tr>
<td>ALTER FUNCTION</td>
<td>Modifies the definition of a function in the dictionary</td>
</tr>
<tr>
<td>Statement</td>
<td>Purpose</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ALTER INDEX</td>
<td>Enables the maximum number of entries to be changed without affecting the existing index structure</td>
</tr>
<tr>
<td>ALTER PROCEDURE</td>
<td>Modifies the definition of a procedure in the dictionary</td>
</tr>
<tr>
<td>ALTER TABLE PROCEDURE</td>
<td>Modifies the definition of a table procedure in the dictionary</td>
</tr>
<tr>
<td>ALTER SCHEMA</td>
<td>Modifies the definition of a schema in the dictionary</td>
</tr>
<tr>
<td>ALTER TABLE</td>
<td>Modifies the definition of a base table in the dictionary</td>
</tr>
<tr>
<td>CREATE CALC</td>
<td>Defines a CALC key on a base table</td>
</tr>
<tr>
<td>CREATE CONSTRAINT</td>
<td>Defines a constraint in the dictionary</td>
</tr>
<tr>
<td>CREATE FUNCTION</td>
<td>Defines a function in the dictionary</td>
</tr>
<tr>
<td>CREATE INDEX</td>
<td>Defines an index on a base table</td>
</tr>
<tr>
<td>CREATE PROCEDURE</td>
<td>Defines a procedure in the dictionary</td>
</tr>
<tr>
<td>CREATE SCHEMA</td>
<td>Defines a schema in the dictionary</td>
</tr>
<tr>
<td>CREATE TABLE</td>
<td>Defines a table in the dictionary</td>
</tr>
<tr>
<td>CREATE TABLE PROCEDURE</td>
<td>Defines a table procedure in the dictionary</td>
</tr>
<tr>
<td>CREATE TEMPORARY TABLE</td>
<td>Defines a temporary table</td>
</tr>
<tr>
<td>CREATE VIEW</td>
<td>Defines a view in the dictionary</td>
</tr>
<tr>
<td>DROP CONSTRAINT</td>
<td>Deletes the definition of a constraint from the dictionary</td>
</tr>
<tr>
<td>DROP CALC</td>
<td>Deletes the definition of a CALC key from the dictionary</td>
</tr>
<tr>
<td>DROP FUNCTION</td>
<td>Deletes the definition of a function from the dictionary</td>
</tr>
<tr>
<td>DROP INDEX</td>
<td>Deletes the definition of an index from the dictionary</td>
</tr>
<tr>
<td>DROP PROCEDURE</td>
<td>Deletes the definition of a procedure in the dictionary</td>
</tr>
<tr>
<td>DROP SCHEMA</td>
<td>Deletes the definition of a schema from the dictionary</td>
</tr>
<tr>
<td>DROP TABLE</td>
<td>Deletes the definition of a base table from the dictionary</td>
</tr>
</tbody>
</table>
## Statement Categories

### Chapter 8: Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROP TABLE PROCEDURE</td>
<td>Deletes the definition of a table procedure in the dictionary</td>
</tr>
<tr>
<td>DROP VIEW</td>
<td>Deletes the definition of a view from the dictionary</td>
</tr>
</tbody>
</table>

### Data Manipulation Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSE*</td>
<td>Places a specified cursor in the closed state</td>
</tr>
<tr>
<td>DECLARE CURSOR*</td>
<td>Defines a cursor for a specified result table</td>
</tr>
<tr>
<td>DECLARE EXTERNAL CURSOR*</td>
<td>Identifies an externally defined global cursor to be used by the application program</td>
</tr>
<tr>
<td>DELETE</td>
<td>Deletes one or more rows from a table</td>
</tr>
<tr>
<td>FETCH*</td>
<td>Retrieves values from the result table associated with a cursor</td>
</tr>
<tr>
<td>INSERT</td>
<td>Adds one or more new rows to a table</td>
</tr>
<tr>
<td>OPEN*</td>
<td>Places a specified cursor in the open state</td>
</tr>
<tr>
<td>SELECT</td>
<td>Retrieves values from one or more tables and views</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Modifies the values in one or more rows of a table</td>
</tr>
</tbody>
</table>

*Programmatic only

### Diagnostics and Statistics Statements

The SQL Diagnostic statements category is used for diagnosing the execution of SQL statements and for returning statistical information for the current transaction.

**Note:** These statements can be used as embedded SQL, including embedding in an SQL-invoked routine. The GET STATISTICS statement can also be used in the SQL command facility and the CA IDMS Visual DBA command console.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET DIAGNOSTICS*</td>
<td>Diagnoses the execution of the last executed SQL statement.</td>
</tr>
</tbody>
</table>
### Statement Categories

<table>
<thead>
<tr>
<th>Statement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET STATISTICS</td>
<td>Returns statistical information for the current transaction.</td>
</tr>
</tbody>
</table>

*Programmatic only

### Dynamic Compilation Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLOCATE CURSOR*</td>
<td>Defines a cursor for a dynamically-prepared statement</td>
</tr>
<tr>
<td>DEALLOCATE PREPARE*</td>
<td>Destroys a dynamically-compiled statement and all other dynamically-compiled statements that directly or indirectly reference it.</td>
</tr>
<tr>
<td>DESCRIBE*</td>
<td>Directs CA IDMS to return information about a dynamically-compiled SQL statement in an SQL descriptor area</td>
</tr>
<tr>
<td>EXECUTE*</td>
<td>Executes a dynamically-compiled SQL statement</td>
</tr>
<tr>
<td>EXECUTE IMMEDIATE*</td>
<td>Dynamically compiles and executes an SQL statement</td>
</tr>
<tr>
<td>PREPARE*</td>
<td>Dynamically compiles an SQL statement for later execution in the application program</td>
</tr>
</tbody>
</table>

*Programmatic only

### Precompiler-directive Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN DECLARE SECTION*</td>
<td>Notifies the precompiler that a host variable definition is beginning.</td>
</tr>
<tr>
<td>END DECLARE SECTION*</td>
<td>Notifies the precompiler that a host variable definition has ended.</td>
</tr>
<tr>
<td>INCLUDE*</td>
<td>Directs the precompiler to create host variable definitions for a specified structure or table in the application program</td>
</tr>
</tbody>
</table>
### Statement Categories

#### Chapter 8: Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHENEVER*</td>
<td>Specifies an action to be taken when the execution of an SQL statement results in a nonzero SQLCODE value</td>
</tr>
</tbody>
</table>

*Programmatic only

### Session Management Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONNECT</td>
<td>Establishes a connection to a CA IDMS dictionary and begins an SQL session</td>
</tr>
<tr>
<td>RELEASE</td>
<td>Releases a connection to a CA IDMS dictionary and ends the SQL session</td>
</tr>
<tr>
<td>RESUME SESSION</td>
<td>Resumes a suspended SQL session</td>
</tr>
<tr>
<td>SET SESSION</td>
<td>Establishes SQL session characteristics</td>
</tr>
<tr>
<td>SUSPEND SESSION</td>
<td>Suspends an SQL session and any transaction currently active within the session</td>
</tr>
</tbody>
</table>

### Transaction Management Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMIT</td>
<td>Makes permanent the changes to the database made during the current transaction and optionally ends the transaction</td>
</tr>
<tr>
<td>ROLLBACK</td>
<td>Cancels changes made to the database during the current transaction and ends the transaction</td>
</tr>
<tr>
<td>SET ACCESS MODULE*</td>
<td>Identifies the access module to be used by a transaction</td>
</tr>
<tr>
<td>SET TRANSACTION</td>
<td>Overrides access module defaults for conditions under which a transaction executes</td>
</tr>
</tbody>
</table>

*Programmatic only
The ALLOCATE CURSOR statement defines a cursor for a dynamically-prepared statement or for a result set returned from a previously invoked procedure.

**Syntax**

```
ALLOCATE extended-cursor-name
CURSOR WITH RETURN FOR extended-statement-name
CURSOR WITHOUT RETURN FOR PROCEDURE SPECIFIC PROCEDURE spec-routine-designator
```

**Parameters**

- **extended-cursor-name**
  
  Identifies the name of the cursor being defined. The name must conform to the rules for an identifier and must be unique within the specified scope.

- **extended-statement-name**
  
  Identifies the name of the statement for which the cursor is being defined. A statement with this name and scope must have been prepared within the same SQL transaction as that in which the ALLOCATE CURSOR statement is being executed.

- **WITH RETURN**
  
  Defines the cursor as a returnable cursor. If a returnable cursor is allocated in an SQL-invoked procedure and is in the open state when the procedure terminates, a result set is returned to the caller.

- **WITHOUT RETURN**
  
  Specifies that the cursor is not a returnable cursor. This is the default.

- **FOR PROCEDURE SPECIFIC PROCEDURE**
  
  Specifies that the cursor is to be allocated for a result set returned by the invocation of the identified procedure. This type of cursor is called a received cursor.

- **spec-routine-designator**
  
  Identifies the SQL-invoked procedure.

**Parameters for Expansion of spec-routine-designator**

- **schema-name**
  
  Specifies the schema with which the procedure identified by `procedure-identifier` is associated.

- **procedure-identifier**
  
  Identifies a procedure defined in the dictionary.
host-variable

Identifies a host variable containing the name of the previously invoked procedure.

routine-parameter

Identifies a routine parameter containing the name of the previously invoked procedure.

local-variable

Identifies a local variable containing the name of the previously invoked procedure.

SCHEMA

Qualifies the procedure name with the name of the schema with which it is associated. This option is an extension to the SQL standard.

schema-name

Specifies the schema with which the procedure is associated.

host-variable

Identifies a host variable containing the name of the schema with which the previously invoked procedure is associated.

routine-parameter

Identifies a routine parameter containing the name of the schema with which the previously invoked procedure is associated.

local-variable

Identifies a local variable containing the name of the schema with which the previously invoked procedure is associated.

Note: For more information about using a schema name to qualify a procedure, see Identifying Entities in Schemas.

Usage

Updateable Cursors

The PREPAREd statement referenced in the ALLOCATE CURSOR statement must be a cursor-specification. The cursor created as a result of the ALLOCATE CURSOR statement, is updateable, if the cursor-specification is updateable.

Allocating a Received Cursor for a Result Set

If the ALLOCATE statement is used for a result set, then the procedure identified by spec-routine-designator must have been previously invoked by an SQL CALL or SELECT statement in the same transaction as that in which the ALLOCATE CURSOR statement is executed.
The result sets that the SQL-invoked procedure returns, form a list ordered in the sequence in which the cursors were opened by the procedure. When a received cursor is allocated, the following actions are taken:

- The new cursor is associated with the first result set in the list of returned result sets.
- The result set is removed from the list.
- The cursor is placed in the open state.
- The cursor is positioned at the same point at which the corresponding returnable cursor was left by the procedure.

If an SQL-invoked procedure has started multiple sessions, the sequence of returned result sets is by session, in the order in which the sessions were connected. Within each session, the result sets are sequenced by the order in which their cursors were opened.

A received cursor cannot be used to return a result set nor can it be referenced in a positioned update or delete statement.

**Note:** For more information about updateable cursors, see DESCRIBE.

**Examples**

**Creating a Local Cursor**

The following ALLOCATE CURSOR statement creates a local cursor called C1 and associates it with the local statement whose name is passed in :sname:

```sql
EXEC SQL
  ALLOCATE 'C1' CURSOR FOR :SNAME
END-EXEC
```

**Creating a Global Cursor**

The following ALLOCATE CURSOR statement creates a global cursor whose name is passed in :CNAME and associates it with the global statement whose name is passed in :SNAME:

```sql
EXEC SQL
  ALLOCATE GLOBAL :CNAME CURSOR FOR :SNAME
END-EXEC
```
Sharing a Statement Definition

The following two ALLOCATE CURSOR statements create two cursors, one of which is local and one of which is global. They are both associated with the same local statement:

```
EXEC SQL
  ALLOCATE 'C1' CURSOR FOR 'S1'
END-EXEC
EXEC SQL
  ALLOCATE GLOBAL CURSOR 'G1' FOR 'S1'
END-EXEC
```

Allocating a Received Cursor for a Result Set

```
EXEC SQL
  call GET_EMPLOYEE_INFO(1003)
END-EXEC
EXEC SQL
  allocate 'RECEIVED_CURSOR_GET_EMPG' for procedure specific procedure GET_EMPLOYEE_INFO
END-EXEC
```

ALTER ACCESS MODULE

The ALTER ACCESS MODULE access module management statement modifies an access module in the dictionary. It is a CA IDMS extension of the SQL standard.

Authorization

To issue an ALTER ACCESS MODULE statement, you must hold the ALTER privilege on or own the access module named in the statement.

In addition to enforcing this authorization requirement, CA IDMS validates the access module owner's authority to execute each DML statement if the dictionary to which the SQL session is connected is controlled by CA IDMS internal security.

If the access module owner does not hold the authority to execute a DML statement in the access module, when the access module is altered, a warning is issued. If the owner still lacks a necessary authority when the access module is executed, an error is returned.
Syntax

```
ALTER ACCESS MODULE access-module-name

VERSION am-version-number

ADD rcm-specification

DROP rcm-name

REPLACE rcm-specification

AUTO RECREATE ON OFF

VALIDATE BY STATEMENT BY MODULE ALL

READ ONLY
READ WRITE

DEFAULT ISOLATION CURSOR STABILITY TRANSIENT READ

READY segment-name.area-name ready-options

Expansion of rcm-specification

```

```
dictionary-name. rcm-name VERSION rcm-version-number
```

Expansion of ready-options

```
SHARED RETRIEVAL
SHARED UPDATE
PROTECTED RETRIEVAL
PROTECTED UPDATE
EXCLUSIVE

INCREMENTAL
PRECLAIM
```
Parameters

_access-module-name_

Specifies the name of the access module being modified. _Access-module-name_ must identify an access module defined and stored in the dictionary.

_schema-name_

Specifies the schema associated with the access module. _Schema-name_ must identify the schema associated with the version of the access module being modified.

If you do not specify _schema-name_, the value used by CA IDMS is the current schema for your SQL session.

_am-version-number_

Specifies the version of the access module to be modified.

If you do not specify _am-version-number_, the version number is set to that found as a result of loading the access module from the dictionary. This depends on the test version number and the loadlist in effect for your user session.

**ADD rcm-specification**

Specifies one or more RCMs to be added to the access module.

Expanded syntax for _rcm-specification_ appears at the end of the statement syntax. Descriptions for these parameters are located at the end of this section.

**DROP rcm-name**

Specifies one or more RCMs to be deleted from the access module.

**REPLACE rcm-specification**

Directs CA IDMS to replace one or more RCMs in the access module with the most recent copies from the dictionary.

Expanded syntax for _rcm-specification_ appears at the end of the statement syntax. Descriptions for these parameters are located at the end of this section.
ALTER ACCESS MODULE

CHANGED

Directs CA IDMS to replace all RCMs whose definition timestamp in the access module does not match the definition timestamp in the RCM load module.

ALL

Directs CA IDMS to recompile all RCMs in the access module.

MAP

Specifies one or more mappings for schema names that qualify table and view identifiers in data manipulation statements. MAP can be specified only with REPLACE ALL.

If you specify MAP, you must supply all schema mappings because existing rules are deleted from the access module.

If you do not specify MAP, schema-name mappings in the existing access module remain in effect.

schema-name-1

Directs CA IDMS to replace occurrences of the specified schema name with the schema name specified in the TO parameter.

NULL

Directs CA IDMS to use the schema name specified in the TO parameter as the qualifier for unqualified table and view identifiers.

TO schema-name-2

Directs CA IDMS to use the specified schema name as the replacement for schema-name-1 or as the qualifier for unqualified table and view identifiers.

AUTO RECREATE

Specifies whether CA IDMS is to re-create the access module after detecting any of the following at runtime:

- An attempt to execute an uncompiled statement
- A change to the definition of a table referenced in the access module
- The execution of a program that has been recompiled since its RCM was included in the access module

CA IDMS identifies the above conditions by comparing definition timestamps in the access module to corresponding timestamps in the database and the host program.

If AUTO RECREATE is not specified, the existing AUTO RECREATE specification for the access module remains in effect.

Note: For more information about the ON and OFF options of AUTO RECREATE, see CREATE ACCESS MODULE.
VALIDATE

Indicates when CA IDMS is to check the definition timestamps of tables in the access module to ensure that the definition has not changed since the access module was created or last altered.

If VALIDATE is not specified, the existing VALIDATE specification for the access module remains in effect.

Note: For more information about the BY STATEMENT, BY MODULE, and ALL options of VALIDATE, see CREATE ACCESS MODULE.

READ ONLY

Specifies transactions started by the access module that do not execute a SET TRANSACTION statement specifying READ WRITE can retrieve data but cannot update the database.

READ WRITE

Specifies transactions started by the access module that do not execute a SET TRANSACTION statement specifying READ ONLY can retrieve data and update the database.

Note: For more information about the READ ONLY and READ WRITE transaction states, see CREATE ACCESS MODULE.

DEFAULT ISOLATION

Specifies the isolation level of transactions started by the access module that do not execute a SET TRANSACTION statement specifying an isolation level.

At runtime, the isolation level of a transaction determines the length of time retrieval locks are held for the purpose of insulating the transaction from the effects of other concurrent transactions. (Update locks are always held until a transaction is committed or rolled back.)

Note: For more information about the CURSOR STABILITY, and TRANSIENT READ DEFAULT ISOLATION options, see CREATE ACCESS MODULE.
READY

Specifies a ready mode for one or more areas accessed through the access module, and specifies when the ready occurs.

The ready mode associated with an area determines:

- Under the central version, the ready mode in which transactions access the area. (The ready mode determines the types of area and row locks CA IDMS places for a transaction.)
- In local mode, the type of physical lock CA IDMS places on the area.

If READY is not specified, the default ready options for areas used by the access module are:

- The existing specifications for areas included in the existing access module
- SHARED UPDATE and INCREMENTAL for areas added as a result of new or replaced RCMs

Parameters for Expansion of rcm-specification

dictionary-name

Identifies the dictionary in which the named RCM is located.

If you do not specify dictionary-name, it is set to the name of the dictionary to which your SQL session is connected.

rcm-name

Identifies the RCM.

Rcm-name must identify an RCM stored in the dictionary and must be unique within the list of RCM names.

rcm-version-number

Identifies the version of the RCM.

If you do not specify rcm-version-number:

1. CA IDMS looks for an RCM with a version number that matches am-version-number
2. If no such RCM is found, CA IDMS looks for version 1
3. If CA IDMS does not find a match, it issues a warning

Parameters for Expansion of ready-options

Note: For more information about ready-options, see CREATE ACCESS MODULE.
Usage

Defaulting the Access Module Version Number

If the version of an access module is not specified, it defaults to the version located as a result of a load operation. This is the same version that would be loaded as a result of executing a program associated with the access module.

For example, assume you have set a test version of 10 and you are using the default loadlist that CA IDMS supplied. CA IDMS loads version 10 of the access module if it exists; otherwise, it loads version 1.

Replacing All or Changed RCMs

When replacing all RCMs in an access module or replacing all RCMs which have been changed since being included in the access module, CA IDMS locates the replacement RCM using the same rules as when the RCM was added to (or explicitly replaced in) the access module. Specifically:

- The dictionary name is the name of the dictionary from which the RCM was previously loaded
- The version is that specified when the RCM was included in the access module, or, if not specified, CA IDMS first looks for an RCM whose version is the same as that of the access module being altered (and if that version is not found, then version 1 of the RCM).
**Dropping RCMs**

When dropping RCMs from an access module, the newly generated access module has a less than optimal structure unless the ALTER statement contains the REPLACE ALL clause. All RCMs need processing to determine the minimum set of control blocks in the access module.

**Avoiding Deadlocks**

If you use the access module of an ALTER ACCESS MODULE statement in other SQL statements in the same session, the ALTER should immediately be followed implicitly or explicitly by a COMMIT. This allows the new copy of the access module to load. Without the COMMIT, a deadlock may occur, even if the two SQL statements refer to different access modules.

**Transaction State and Isolation Level**

If you specify neither transaction state nor DEFAULT ISOLATION on an ALTER ACCESS MODULE statement, the existing values remain in effect. If either is specified, it also establishes a value for the other, as follows:

- If READ WRITE or READ ONLY are specified, CURSOR STABILITY is assumed
- If TRANSIENT READ is specified, READ ONLY is assumed
- If CURSOR STABILITY is specified, READ WRITE is assumed

**Examples**

**Replacing Changed RCMs**

The following ALTER ACCESS MODULE statement replaces any changed RCMs in access module EMPAM001 with the most recent copies from the dictionary:

```
alter access module hrprod.empam001
    replace changed;
```

**Adding New RCMs**

The following ALTER ACCESS MODULE statement adds two new RCMs to the SALES001 access module. The statement also changes the lock options for two areas.

```
alter access module prod.sales001
    add sales.bdgt_001,
    add sales.comm_003
    ready
        salesseg.sales_area shared update incremental
demoseg.emp_area shared retrieval preclaim;
```
ALTER CATALOG

The ALTER CATALOG statement establishes the character set for character columns of the tables defined within the current dictionary. The character set is used to correctly sort additional national characters for the named language. It is also a CA IDMS extension of the SQL standard.

Authorization

To issue an ALTER CATALOG statement, you must have DBADMIN authority, and ALTER authority for the SYSTEM schema.

Note: This statement must be executed immediately following the execution of the TABLEDDL file that records the definition of the catalog tables themselves. The statement must precede the definition of any user SQL tables in that catalog.

Syntax

```
ALTER CATALOG DEFAULT CHARACTER SET alternate-character-set-name
```

Parameters

*alternate-character-set-name*

Identifies the character set whose collating scheme is to be used for the CHAR and VARCHAR columns defined in the catalog. Currently-supported values for *alternate-character-set-name* are DENMARK, FINLAND, NORWAY, and SWEDEN.
Usage

The ALTER CATALOG DEFAULT CHARACTER SET statement changes the collating sequence for all CHAR and VARCHAR columns defined in user tables in the dictionary, including non-SQL defined tables. It does not affect columns in the SYSTEM tables.

Swedish and Finnish

The additional national characters, the uppercase forms of which are represented by the symbols $, #, and &theta., sort at the end of the standard alphabet. The accented E and U sort the same as their unaccented equivalents, and are returned to application programs as their unaccented equivalents.

Norwegian and Danish

The additional national characters, the uppercase form of which is represented by the symbols #, &theta., and $, sort at the end of the standard alphabet. The accented U sort in the same sequence as its unaccented equivalent and are returned to application programs as an unaccented U.

Example

The following statement causes the data values of all CHAR and VARCHAR columns to collate according to the conventions of the Finnish alphabet.

alter catalog default character set FINLAND

ALTER CONSTRAINT

The ALTER CONSTRAINT statement changes the characteristics of an existing referential constraint. This statement is a CA IDMS extension to the SQL standard.

Authorization

To issue an ALTER CONSTRAINT statement, you must:

- Either hold the ALTER privilege on or own the referencing table in the constraint being altered
- Hold the REFERENCES privilege on the referenced table in the constraint being altered

Note: To issue an ALTER CONSTRAINT statement you must own or hold the ALTER privilege on the table on which the constraint is defined.
ALTER CONSTRAINT

Syntax: ALTER CONSTRAINT

```
ALTER CONSTRAINT constraint-name ON referencing-table
```

**INDEX BLOCK CONTAINS** key-count KEYS

**DISPLACEMENT IS** page-count PAGES

**NOT** UNIQUE

**ALTER CONSTRAINT Parameters**

This section describes the ALTER CONSTRAINT parameters:

**constraint-name**

Identifies the referential constraint to be changed. Constraint-name must be the name of a constraint on the table identified in the ON clause.

**referencing-table**

Specifies the name of the referencing table in the constraint to be changed.

**schema-name**

Identifies the schema associated with the referencing table.

**Default:** The default varies depending on where the statement is encountered.

If you do not specify schema-name, it defaults to:

- The current schema associated with your SQL session when the statement is entered through the Command Facility or executed dynamically.

- The schema associated with the access module used at runtime when the statement is embedded in an application program.

**key-count KEYS**

Establishes a new value for the maximum number of entries in each internal index record (SR8 system record).

**Limits:** Key-count must be an unsigned integer in the range 3 through 8180.

**page-count PAGES**

Specifies how far away from the referenced row the bottom-level index records are stored.

If the value of page-count is zero (0), the bottom-level internal index records are not displaced from the referenced row.

**Limits:** Page-count must be an unsigned integer in the range 0 through 32,767.
UNIQUE

Specifies that the sort-key value in any given row of the referencing table must be different from the sort-key value in all other rows that have the same non-null referencing key value.

NOT UNIQUE

Removes the restriction that all values of the sort-key with the same non-null foreign key value must be unique.

Example: Alter the DEPT_EMPL Constraint

In this example, the physical characteristics of the DEPT_EMPL constraint are changed. Each internal index record will have a maximum of 10 keys and the bottom level index records will be displaced 50 pages from the associated referenced row:

```
alter constraint dept_empl on emp.empl
    displacement is 50 pages
    index block contains 10 keys;
```

ALTER FUNCTION

The ALTER FUNCTION data description statement modifies the definition of a function in the dictionary.

Using the ALTER FUNCTION statement, you can:

- Revise the estimated row and I/O counts
- Change the external name of the function
- Change the size and characteristics of the work areas passed to the function
- Change the execution mode of the function
- Change the protocol
- Change the language of the function
- Change the timestamp
- Change the default database
- Change the transaction sharing mode

The ability to change attributes other than language and external name is a CA IDMS extension of the SQL standard.

Authorization

To issue an ALTER FUNCTION statement, you must either own or hold the ALTER privilege on the function named in the statement.
### Syntax

```
ALTER FUNCTION function-identifier
    schema-name.

EXTERNAL NAME external-routine-name

ESTIMATED ROWS row-count
ESTIMATED IOS io-count
LOCAL WORK AREA local-stge-size
GLOBAL WORK AREA global-stge-size

USER MODE
SYSTEM MODE
PROTOCOL language-clause

timestamp-value

DEFAULT DATABASE

key-ID

NULL

TRANSACTION SHARING
```

**Expansion of language-clause**

```
LANGUAGE
```

**Parameters**

- **function-identifier**
  Specifies the name of the function being modified. `function-identifier` must identify a function defined in the dictionary.

- **schema-name**
  Identifies the schema associated with the named function.
  If you do not specify a `schema-name`, the default value is:
  - The current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically.
  - The SQL schema associated with the access module used at runtime, if the statement is embedded in an application program.

- **external-routine-name**
  Specifies the one- to eight-character name of the program which CA IDMS calls to process function invocations.
row-count

Specifies an integer value, in the range of 0 through 2,147,483,647, which represents the average number of rows that the CA IDMS optimizer uses for cost calculation of the function invocation.

io-count

Specifies an integer value, in the range of 0 through 2,147,483,647, which represents the average number of disk accesses that the function generates for a given set of input parameters.

local-stge-size

Specifies an integer, in the range of 0 through 32767, which represents the size, in bytes, of a local storage area that CA IDMS allocates at runtime and passes to the function on each invocation.

CA IDMS allocates a local storage area on the first call to a function.

global-stge-size

Specifies an integer, in the range of 0 through 32767, which represents the size, in bytes, of a global storage area that CA IDMS allocates at runtime and passes to the function on each invocation.

CA IDMS allocates a global storage area once within a transaction and retains it until the transaction terminates.

key-id

Specifies the one- to four-character identifier for the global storage area. CA IDMS passes the same piece of global storage within a transaction to all routines that have the same global storage key.

If you do not specify a storage key, its value remains unchanged. To remove a storage key, specify NULL as the key.

USER MODE

Specifies that the function should execute as a user-mode application program within CA IDMS. Do not specify user mode for functions specified with protocol ADS, such as is the case with functions written as CA ADS mapless dialogs or written in SQL.

SYSTEM MODE

Specifies that the procedure should execute as a system-mode application program. To execute as a system mode application, the program must be fully reentrant and written in either:

■ ADS as a mapless dialog
■ SQL
■ Assembler using DC calling conventions
■ COBOL or PL/I and compiled with an LE-compliant compiler
PROTOCOL

Specifies the environment.

IDMS

Use IDMS for SQL-invoked functions that are written in COBOL, PL/I, or Assembler.

ADS

Use ADS for SQL-invoked functions that are written in SQL or CA ADS. The name of the dialog that will be loaded and run when the SQL function is invoked is given by the external-routine-name in the EXTERNAL NAME clause. Setting the protocol to ADS, requires the function to have its mode set to system.

language-clause

Specifies the programming language of the function.

timestamp-value

Specifies the value of the synchronization stamp to be assigned to the function. Timestamp-value must be a valid external representation of a timestamp.

DEFAULT DATABASE

Specifies whether a default database should be established for database sessions started by the function.

NULL

Specifies that no default database should be established.

CURRENT

Specifies that the database to which the SQL session is connected should become the default for any database session started by the function.

TRANSACTION SHARING

Specifies whether to enable transaction sharing for database sessions started by the function. If transaction sharing is enabled for a function's database session, it will share the current SQL session's transaction.

ON

Specifies to enable transaction sharing.

OFF

Specifies to disable transaction sharing.

DEFAULT

Specifies to retain the transaction sharing setting that is in effect when the function is invoked.
Parameters for Expansion of language-clause

**ADS**
Specifies that the SQL routine is written in the CA ADS language.

**ASSEMBLER**
Specifies that the SQL routine is written in the assembler language.

**COBOL**
Specifies that the SQL routine is written in the COBOL language.

**PLI**
Specifies that the SQL routine is written in the PL/I language.

**SQL**
Specifies that the SQL routine is written in the SQL language.

**Note:** The ability to specify ADS or ASSEMBLER as a language is a CA IDMS extension.

**Usage**

**Changing the language of a function**

A function with language SQL cannot be changed to any other language and a function whose language is not SQL cannot be changed to language SQL.

**Specifying a Synchronization Stamp**

When defining or altering a function, you can specify a value for its synchronization stamp. You should use care when doing so because the purpose of the stamp is to enable the detection of discrepancies between an entity and its definition. If explicitly specified, you must set the synchronization stamp to a new value following a change so that the change is detectable by the runtime system.

If not specified, the synchronization stamp is automatically set to the current date and time.

**Note:** For more information about creating a function, see CREATE FUNCTION.

**Example**

The following example shows the use of ALTER FUNCTION to change the external name of a function.

```
alter function fin.udf_funbonus external name funbon09;
```
ALTER INDEX

The ALTER INDEX statement alters the characteristics of an existing index. It is also a CA IDMS extension of the SQL standard. You can change the structure and location of an index through the modification of the following attributes:

- Key count
- Displacement
- Uniqueness
- Area association

Authorization

To issue an ALTER INDEX statement, you must have the ALTER privilege on or own the table on which the index is defined.

Syntax

```sql
ALTER INDEX index-name ON table-identifier
```

Parameters

This section describes the parameters for the ALTER INDEX statement:

**page-count PAGES**

Specifies how far away from the index owner the bottom-level index records are stored.

- If the value of page-count is zero (0), the bottom-level internal index records are not displaced from the index owner.

  Limit: An unsigned integer from 0–32,767.

**UNIQUE**

Specifies that the index-key value in any given row of the table on which the index is defined must be different from the index-key value in all other rows of the table. The table cannot contain any duplicate index-key values.

- If you specify UNIQUE and the table contains duplicate index-key values, the alter statement will fail.
NOT UNIQUE

Removes the restriction that all values of the index-key within the table must be unique.

When the UNIQUE restriction is used to ensure uniqueness of a referenced key in some constraint, you cannot remove it from an index unless another index or CALC key can be used in its place.

IN

Requests a change in the location of the named index.

area-name

Identifies a new area with which the index is to be associated. Area-name must identify an area defined in the dictionary.

segment-name

Identifies the segment associated with the area.

Usage

System tables

You cannot alter an index defined on a table in the SYSTEM schema.

Changing the Number of Entries in an SR8

It is sometimes desirable to change the number of entries in an SR8 system record after an index has been loaded. The ALTER INDEX statement enables the maximum number of entries to be changed without affecting the existing index structure.

Note: For more information about index structure and design considerations, see the CA IDMS Database Administration Guide.

Example

In this example, the EMP_LNAME index is moved from its current location to the DEMO.EMPAREA area. Each internal index record will have a maximum of 30 keys and the bottom-level index records will be displaced 40 pages from the top of the index.

```
alter index emp_lname (last_name) on emp.benefits
    displacement is 40 pages
    index block contains 30 keys
    in area demo.emparea;
```
ALTER PROCEDURE

The ALTER PROCEDURE data description statement modifies the definition of a procedure in the dictionary. Using the ALTER PROCEDURE statement, you can:

- Add a new parameter to a procedure
- Revise the estimated row and I/O counts
- Change the external name of the procedure
- Change the size and characteristics of the work areas passed to the procedure
- Change the execution mode of the procedure
- Change the language of the procedure
- Update the timestamp
- Change the default database option
- Change the transaction sharing option
- Change the protocol
- Change the maximum number of dynamic result sets

The ability to change attributes other than language, external name, and the maximum number of dynamic result sets is a CA IDMS extension of the SQL standard.

Authorization

To issue an ALTER PROCEDURE statement, you must either own or hold the ALTER privilege on the procedure named in the statement.

Syntax

```
ALTER PROCEDURE schema-name. procedure-identifier
ADD parameter-definition

ADD ( parameter-definition )
EXTERNAL NAME external-routine-name
ESTIMATED ROWS row-count
ESTIMATED I/Os io-count
LOCAL WORK AREA local-stge-size
GLOBAL WORK AREA global-stge-size

USER MODE
SYSTEM MODE
PROTOCOL IDMS ADS

language-clause

TIMESTAMP timestamp-value
DEFAULT DATABASE NULL CURRENT
TRANSACTION SHARING ON OFF DEFAULT
DYNAMIC RESULT SETS maximum-dynamic-result-sets
```
Parameters

procedure-identifier

Specifies the name of the procedure being modified. Procedure-identifier must identify a procedure defined in the dictionary.

schema-name

Identifies the schema associated with the named procedure. If you do not specify a schema-name it defaults to:

- The current schema associated with your SQL session, if you enter the statement through the Command Facility or execute it dynamically
- The schema associated with the access module used at runtime, if the statement is embedded in an application program

parameter-definition

Defines one or more new parameters to be associated with the procedure. New parameters are added, in the order specified, after the last existing parameter.

For a description of parameter-definition, see CREATE PROCEDURE. Descriptions for the expansion parameters are located at the end of this section.
**external-routine-name**

Specifies the one- to eight-character name of the program which CA IDMS calls to process references to the procedure.

**row-count**

Specifies an integer value, in the range of 0 through 2,147,483,647, which represents the average number of rows that the procedure returns for a given set of input parameters.

**io-count**

Specifies an integer value, in the range of 0 through 2,147,483,647, which represents the average number of disk accesses that the procedure generates for a given set of input parameters.

**local-stge-size**

Specifies an integer, in the range of 0 through 32767, which represents the size, in bytes, of a local storage area that CA IDMS allocates at runtime and passes to the procedure on each invocation.

**global-stge-size**

Specifies an integer, in the range of 0 through 32767, which represents the size, in bytes, of a global storage area that CA IDMS allocates at runtime and passes to the procedure on each invocation.

CA IDMS allocates a global storage area once within a transaction and retains it until the transaction terminates.

**key-id**

Specifies the one- to four-character identifier for the global storage area. CA IDMS passes the same piece of global storage within a transaction to all routines that have the same global storage key.

If you do not specify a storage key, its value remains unchanged. To remove a storage key, specify NULL as the key.
USER MODE

Specifies that the procedure should execute as a user-mode application program within CA IDMS. Do not specify user mode for procedures specified with protocol ADS, such as is the case with procedures written as CA ADS mapless dialogs or written in SQL.

SYSTEM MODE

Specifies that the procedure should execute as a system mode application program. To execute as a system mode application, the program must be fully reentrant and be written in either:

- ADS as a mapless dialog
- SQL
- Assembler using DC calling conventions
- COBOL or PL/I and compiled with an LE-compliant compiler

PROTOCOL

Specifies the environment.

IDMS

Use IDMS for SQL-invoked functions that are written in COBOL, PL/I, or Assembler.

ADS

Use ADS for SQL-invoked functions that are written in SQL or CA ADS. The name of the dialog that will be loaded and run when the SQL function is invoked is given by the external-routine-name in the EXTERNAL NAME clause. Setting the protocol to ADS, requires the function to have its mode set to system.

language-clause

Specifies the programming language of the procedure.

timestamp-value

Specifies the value of the synchronization stamp to be assigned to the procedure. Timestamp-value must be a valid external representation of a timestamp.

DEFAULT DATABASE

Specifies whether a default database should be established for database sessions started by the procedure.

NULL

Specifies that no default database should be established.

CURRENT

Specifies that the database to which the SQL session is connected should become the default for any database session started by the procedure.
TRANSACTION SHARING

Specifies whether transaction sharing should be enabled for database sessions started by the procedure. If transaction sharing is enabled for a procedure's database session, it will share the current SQL session's transaction.

ON

Specifies that transaction sharing should be enabled.

OFF

Specifies that transaction sharing should be disabled.

DEFAULT

Specifies that the transaction sharing setting that is in effect when the procedure is invoked should be retained.

DYNAMIC RESULT SETS

Defines the maximum number of result sets that a procedure invocation can return to its caller. A result set is a sequence of rows specified by a cursor-specification, created by the opening of a cursor and ranged over that cursor.

maximum-dynamic-result-sets

Defines an integer in the range 0-32767 specifying the maximum number of result sets a procedure can return.

Parameters for Expansion of parameter-definition

parameter-name

Specifies a 1- to 32-character name of a parameter to be passed to the table procedure. Parameter-name must:

- Be unique within the table procedure that you are defining
- Follow the conventions for SQL identifiers

All parameters are implicitly nullable. Input parameters can be assigned NULL as a parameter value and output parameters can return NULL.

data-type

Defines the data type for the named parameter. For expanded data-type syntax, see Expansion of Data-type.
WITH DEFAULT

Directs CA IDMS to pass a default value for the named parameter if no value for the parameter is specified.

The default value for a parameter is based on its data type:

<table>
<thead>
<tr>
<th>Column data type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>Blanks</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>A character string literal with a length of zero (that is, '')</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>Double-byte blanks</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>A double-byte character string literal with a length of zero</td>
</tr>
<tr>
<td>DATE</td>
<td>The value in the CURRENT DATE special register</td>
</tr>
<tr>
<td>TIME</td>
<td>The value in the CURRENT TIME special register</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>The value in the CURRENT TIMESTAMP special register</td>
</tr>
<tr>
<td>All numeric data types</td>
<td>0 (zero)</td>
</tr>
</tbody>
</table>

Parameters for Expansion of language-clause

ADS
Specifies that the SQL routine is written in the CA ADS language.

ASSEMBLER
Specifies that the SQL routine is written in the assembler language.

COBOL
Specifies that the SQL routine is written in the COBOL language.

PLI
Specifies that the SQL routine is written in the PL/I language.

SQL
Specifies that the SQL routine is written in the SQL language.

Note: The ability to specify ADS or ASSEMBLER as a language is a CA IDMS extension.
Usage

Changing the language of a procedure

A procedure with language SQL cannot be changed to any other language, and a procedure whose language is not SQL cannot be changed to language SQL.

Specifying a Synchronization Stamp

When defining or altering a procedure, you can specify a value for its synchronization stamp. You should use care when doing so because the purpose of the stamp is to enable the detection of discrepancies between an entity and its definition. If explicitly specified, you must set the synchronization stamp to a new value following a change so that the change is detectable by the runtime system.

If not specified, the synchronization stamp is automatically set to the current date and time.

Note: For more information about creating a procedure, see CREATE PROCEDURE.

Examples

Adding Parameters to a Procedure

The following ALTER PROCEDURE statement adds two new parameters to the EMP.GET_BONUS procedure:

```
alter procedure emp.get_bonus
  add (start_month  char (2),
      start_year   char (2));
```

ALTER SCHEMA

The ALTER SCHEMA data description statement that modifies the definition of a schema in the dictionary. It is also a CA IDMS extension of the SQL standard.

Authorization

To issue an ALTER SCHEMA statement, you must hold the ALTER privilege on the schema named in the statement.

If you specify FOR NONSQL SCHEMA, you must have the USE privilege on the non-SQL schema.

If you specify DBNAME, you must have USE privilege on the database; if you do not specify DBNAME or specify a value of NULL, you must have DBADMIN privilege on DBNAME SYSTEM.
**ALTER SCHEMA**

## Syntax

```sql
ALTER SCHEMA schema-name

- DEFAULT AREA segment-name.area-name NULL
- DBNAME database-name NULL
- FOR NONSQL SCHEMA nonsql-schema-specification
- FOR SQL SCHEMA sql-schema-specification
```

### Expansion of nonsql-schema-specification (ALTER SCHEMA)

```sql
dictionary-name. nonsql-schema-name VERSION version-number

- DBNAME database-name
```

### Expansion of sql-schema-specification (ALTER SCHEMA)

```sql
sql-schema-name

- DBNAME database-name
```

## Parameters

### schema-name

Specifies the name of the schema being modified. *Schema-name* must identify a schema defined in the dictionary.

### DEFAULT AREA

Modifies the default area specification for the named schema. This parameter is valid only for a schema that is not associated with a non-SQL-defined schema.

The named area is used by default for storing rows of tables *subsequently* defined in the named schema. It replaces any previous default area specification for the schema.

### segment-name.area-name

Identifies the segment and area.

You do not need to define the named segment or area in the dictionary before issuing the ALTER SCHEMA statement.
NULL

Removes any previous default area specification for the named schema.

If the default area specification is removed, all subsequent CREATE TABLE
statements that qualify the table name with the name of the schema being altered
must include the IN parameter.

DBNAME

If the schema has been associated with a non-SQL-defined schema, you can add or
change the specification of the database using this parameter.

Descriptions of the database-name and NULL parameters are presented under
nonsql-schema-specification.

nonsql-schema-specification

Identifies the non-SQL-defined schema to associate with the SQL schema.

Expanded syntax for nonsql-schema-specification appears immediately following
the statement syntax. Descriptions for these parameters are located at the end of
this section.

sql-schema-specification

Identifies an existing SQL-defined schema to which the new SQL schema refers.
Expanded syntax for sql-schema-specification appears immediately following the
statement syntax.

Parameters for Expansion of nonsql-schema-specification

dictionary-name.

Names the dictionary that contains the non-SQL-defined schema.

If you do not specify dictionary-name, it is set to the dictionary to which your SQL
session is connected.

nonsql-schema-name

Identifies the non-SQL-defined schema.

VERSION version-number

Identifies the version number of the non-SQL-defined schema. If VERSION
version-number is not specified, version-number defaults to 1.
DBNAME

Specifies the database that the non-SQL-defined schema describes or removes a database specification.

For considerations about whether to specify a database when you create a schema for a non-SQL-defined schema, see the "Usage" section of CREATE SCHEMA.

database-name

Identifies one of the following:

- The segment containing the areas described by the non-SQL-defined schema.
- A database name that includes segments containing the areas described by the non-SQL-defined schema.

At runtime CA IDMS accesses the segments associated with the database name that contain areas with the same name as the areas in the non-SQL-defined schema.

NULL

Initializes the database name for the non-SQL-defined schema to blanks.

If no database-name is specified in the schema definition, at runtime the database name to which the SQL session is connected must include segments containing the areas described by the non-SQL-defined schema.

Parameters for Expansion of sql-schema-specification

sql-schema-name

Names the referenced SQL-defined-schema. this named schema must not itself reference another schema.

DBNAME database-name

Identifies the database containing the data described by the referenced SQL-defined schema. Database-name must be a database name that is defined in the database name table or a segment name defined in the DMCL.
Usage

System-owned Schema

You cannot modify the definition of the schema named SYSTEM.

Changing non-SQL-defined Schema Information

If you change the name or version number of the non-SQL defined schema associated with an SQL-defined schema or if you change the database name associated with the schema, you must recompile all affected access modules and drop and recreate all affected views.

To determine which access modules are affected, use the DISPLAY ALL ACCESS MODULE statement with the TABLE selection criteria.

To recompile an affected access module, use the ALTER ACCESS MODULE statement with the REPLACE ALL option.

Views must be dropped and recreated if the structure of one or more referenced records in the new non-SQL-defined schema is different than the structure at the time the view was created. Views are also invalid if a referenced record has been deleted from the non-SQL schema. To determine which views are affected, use the DISPLAY ALL VIEW statement with the REFERENCED selection criteria. Before dropping the view, display its syntax by using the DISPLAY or PUNCH VIEW statement.

Restricted Changes

You cannot alter the type of a schema:

- You cannot change a non-referencing schema to a referencing schema or a referencing schema to a non-referencing schema.
- You cannot change the type of schema being referenced from SQL to non-SQL or from non-SQL to SQL.

Changing Referenced SQL Schema Information

If you change the name of the SQL schema that is referenced, you must drop and recreate all views that reference tables in the referencing schema, for example, the schema being altered. To determine which views are affected, use the DISPLAY ALL VIEW statement with the REFERENCED selection criteria. Before dropping the view, display its syntax by using the DISPLAY or PUNCH VIEW statement.
Example

Removing the Default Area Specification

The following ALTER SCHEMA statement removes the default area specification from the SALES schema:

alter schema sales
default area null;

More Information

- For more information about defining schemas, see CREATE SCHEMA and DROP SCHEMA.
- For more information about non-SQL schemas, see Accessing Non-SQL-Defined Databases.
- For more information about displaying access modules and views, see DISPLAY/PUNCH ACCESS MODULE and DISPLAY/PUNCH VIEW.

ALTER TABLE

The ALTER TABLE data description statement modifies the definition of a base table in the dictionary.

Using ALTER TABLE, you can perform the following tasks:

- Add one or more columns to a table
- Alter a column’s data type or null attribute
- Drop or change a column’s default clause
- Rename a column
- Drop a column
- Specify additional restrictions on the data that can be stored in a table
- Remove all restrictions on the data that can be stored in a table
- Add or delete the default index associated with a table
- Revise the estimated row count for a table
- Update the table’s timestamp

The ability to revise the estimated row count and to update the table’s timestamp is a CA IDMS extension of the SQL standard.
Authorization

To issue an ALTER TABLE statement, you must hold the ALTER privilege on or own the table named in the statement.

Syntax

```sql
ALTER TABLE schema-name table-identifier
```

```
ADD CHECK ( search-condition )
DROP CHECK
ADD DEFAULT INDEX
DROP DEFAULT INDEX
ADD COLUMN column-definition
ADD (column-definition)
DROP COLUMN column-alteration
ALTER COLUMN column-name column-definition
RENAME COLUMN column-name TO new-column-name
ESTIMATED ROWS estimated-row-count
TIMESTAMP timestamp-value
```

Expansion of column-definition

```sql
column-name data-type NOT NULL WITH DEFAULT
```

Expansion of column-alteration

```sql
column-name SET DATA TYPE data-type ALLOW NULL NOT WITH DEFAULT DROP DEFAULT
```

Parameters

**table-identifier**

Specifies the name of the table being modified. *Table-identifier* must identify a base table defined in the dictionary.

**schema-name**

Identifies the schema associated with the named table.

If not specified, *schema-name* defaults to:

- The current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically
- The schema associated with the access module used at runtime, if the statement is embedded in an application program
column-definition

Defines one or more new columns to be included in the table. New columns are added after the last existing column.

Expanded syntax for column-definition is shown immediately following the ALTER TABLE syntax. Descriptions of column-definition parameters follow description of ALTER TABLE parameters.

ADD CHECK (search-condition)

Specifies additional restrictions on the data that can be stored in the table.

If the table definition already includes data restrictions in a search condition, CA IDMS appends the search condition specified in the ADD CHECK parameter to the existing search condition with the binary operator AND. CA IDMS stores a new row in the table only if the value of the entire expression formed by the concatenation of the search conditions is true.

Restrictions on the use of search-condition with ADD CHECK are discussed under "Usage" following these parameter descriptions. For expanded search-condition syntax, see Expansion of Search-condition.

DROP CHECK

Removes any existing restrictions on the data that can be stored in the table.

ADD DEFAULT INDEX

Creates a default index for the named table.

Note: The table must not have a default index already associated with it.

DROP DEFAULT INDEX

Deletes the default index associated with the table.

column-alteration

Specifies the changes to be made to the attributes of a column.

Note: The expanded syntax for column-alteration is shown after the ALTER TABLE syntax. Descriptions of column-alteration parameters follow the description of ALTER TABLE parameters.

DROP COLUMN column-name

Identifies the column to be removed from the table. Column-name must be the name of a column in the table.

Note: You cannot drop columns that are part of a CALC key of a populated table or that are named in a check constraint.
CASCADE

Drops the following entities:

■ The CALC key if it includes the column.
■ All referential constraints in which the named column is a referenced or a foreign key column.
■ All linked constraints in which the named column is a sort column.
■ All indexes in which the named column is an indexed column.
■ All views in which the column is named.

Note: If CASCADE is not specified, the column must not participate in a referential constraint or index, or be named in a view.

RENAME COLUMN column-name

Identifies the column name to be changed. Column-name must be the name of a column in the table.

Note: You cannot rename a column if the column is named in a check constraint or in a view.

TO new-column-name

Specifies the new name for the identified column.

Limit: 1–32 characters that follows the SQL identifier standard.

Note: The new column name must be distinct from the name of any existing column in the table.

ESTIMATED ROWS estimated-row-count

Indicates the number of rows expected to be stored for the table. Estimated-row-count must be an integer that does not exceed 16,777,214. The specified value replaces any previous estimated row count for the table.

TIMESTAMP timestamp-value

Specifies the value of the synchronization stamp to be assigned to the table. Timestamp-value must be a valid external representation of a timestamp.

Parameters for Expansion of column-definition

column-name

Specifies the name of a column to be included in the table being created. Column-name must be a one- through 32-character name that follows the conventions for SQL identifiers. Column-name must be unique within the table being defined.

data-type

Defines the data type for the named column. For expanded data-type syntax, see Expansion of Data-type.
NOT NULL

Indicates that the column cannot contain null values.

If NOT NULL is specified without WITH DEFAULT, the table being altered must be empty.

If NOT NULL is not specified, the column is defined to allow null values.

WITH DEFAULT

Directs CA IDMS to establish a default value for the column being added.

The default value is based on the data type of the column:

<table>
<thead>
<tr>
<th>Column data type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>Blanks</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>A character string literal with a length of zero (that is, &quot;&quot;)</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>Double-byte blanks</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>A double-byte character string literal with a length of zero</td>
</tr>
<tr>
<td>DATE</td>
<td>'0001-01-01' for existing rows</td>
</tr>
<tr>
<td></td>
<td>The value in the CURRENT DATE special register for newly inserted rows</td>
</tr>
<tr>
<td>TIME</td>
<td>'00.00.00' for existing rows</td>
</tr>
<tr>
<td></td>
<td>The value in the CURRENT TIME special register for newly inserted rows</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>'0001-01-01-00.00.00.000000' for existing rows</td>
</tr>
<tr>
<td></td>
<td>The value in the CURRENT TIMESTAMP special register for newly inserted rows</td>
</tr>
<tr>
<td>All numeric data types</td>
<td>0 (zero)</td>
</tr>
</tbody>
</table>

If you do not specify WITH DEFAULT, then:

- If you specify NOT NULL, the table must be empty
- If you do not specify NOT NULL, the default value for the column is NULL
Parameters for Expansion of column-alteration

column-name
Identifies the column whose attributes are to be changed. Column-name must be the name of a column in the table.

data-type
Defines the new data type for the named column. The specified data type must be compatible for assignment with the column’s existing data type. For expanded data-type syntax, see Expansion of Data-type.
You cannot change the data type of a column that is part of a CALC key of a populated table or that is a referenced or foreign key column in a constraint.

ALLOW NULL
Indicates that the column can contain null values. You cannot change the null attribute of a column that is part of a CALC key of a populated table or a referenced key.

NOT NULL
Indicates that the column cannot contain null values. You cannot change the null attribute of a column that is part of a CALC key of a populated table or a referenced key.

WITH DEFAULT
Sets the column’s value to a default if no value for the column is specified when a row is inserted.

DROP DEFAULT
Does not set the column’s value to a default when a row is inserted.
Usage

Tables in System Schemas

You cannot modify the definition of a table in the SYSTEM schema.

Maximum Row Length

When adding a column to a table, you must ensure that the total number of bytes required for all columns in the table does not exceed the maximum allowed.

Note: For more information about maximum row length, see CREATE TABLE.

Restrictions on search-condition

In the ADD CHECK parameter of an ALTER TABLE statement:

- **Search-condition** cannot include any host variables, local variables, routine parameters, aggregate or user-defined functions, EXISTS predicates, quantified predicates, or subqueries
- Each column reference in **search condition** must identify a column in the table being modified

Modifying Tables that Contain Data

If the table specified in an ALTER TABLE statement contains one or more rows of data (that is, the table is not empty), and the ALTER TABLE statement specifies:

- **ADD column-definition**, you must supply a default value in the DEFAULT parameter of the column definition if you specify NOT NULL
- **ADD CHECK**, the value of the search condition specified in the ADD CHECK parameter must be true for each existing row in the table

Add a Default to a Column

Allowing a column to have a default value affects only the table’s definition; existing table rows are not affected.
Remove a Column’s Default

If the table is populated and the column does not allow null values, every existing row must contain a value in the changed column. To ensure this, each row is accessed and updated if it does not contain a value for the column.

Rename a Column

A column that is named in a check constraint or a view cannot be renamed.

The definition of all referential constraints, sort keys, CALC keys and indexes in which the column participates are updated to show the new column name.

Drop a Column

Every row in the table is updated to remove the column value.

If a column is named in a check constraint or is part of the CALC key of a populated table, you cannot drop the column.

If you do not specify CASCADE, the column must not be one of the following types of columns:

- A column in a CALC key
- A referenced or foreign key column in a referential constraint
- An indexed column
- A sort column of a linked constraint
- Named in a view

If you specify CASCADE, how the column is used determines what other items are dropped:

- Dropping a CALC key column also drops the CALC key
- Dropping a referenced or foreign key column in a referential constraint also drops the constraint
- Dropping an indexed column also drops the index
- Dropping a sort column of a linked constraint also drops the constraint
- Dropping a column named in a view also drops the view
Change a Column’s Null Attribute

The following situations apply when you change a column’s null attribute:

- When the column is part of a CALC key of a populated table, or is a referenced column in a constraint, the ALTER statement fails.
- When you change a null attribute, every row in the table is updated to add or remove the null attribute byte for that column.
- When the changed column is a sort column, every index and linked indexed constraint is automatically rebuilt.
- When disallowing nulls and the value of the column is null for a row in the table, the ALTER statement fails.

Change a Column’s Data Type

The following situations apply when you change a column’s data type:

- When the column is part of a CALC key of a populated table, or is a referenced column in a constraint, the ALTER statement fails.
- When changing a column’s data type, the new data type you enter must be compatible for assignment with the original data type.
- Every row in the table is restructured to convert the column value to the new type. This might involve increasing or decreasing the length of the row.
- The ALTER statement will fail if a loss of data (such as truncation of a non-blank character or numeric overflow) would occur as part of the conversion.
- When you change data type, every index and linked indexed constraint in which the column is a sort column is rebuilt.

Examining Check Constraints on a Table

You can examine the current check constraint on a table by using the DISPLAY TABLE statement.

Examining existing check constraints is useful if you are planning to change a constraint by dropping it and adding the changed constraint.

Adding Columns with Multiple ALTER TABLE Statements

When columns are added with the ALTER TABLE statement, the first column in the column definition list is aligned on a full word boundary in the physical data structure that represents table rows. Since each individual ALTER TABLE statement will cause alignment, columns added in separated ALTERs versus one ALTER can result in different row lengths and column offsets within the physical row data structure.
Specifying a Synchronization Stamp

When defining or altering a table, you can specify a value for its synchronization stamp. You should use care when doing so because the purpose of the stamp is to enable the detection of discrepancies between an entity and its definition. If explicitly specified, you must set the synchronization stamp to a new value following a change so that the change is detectable by the runtime system.

If not specified, the synchronization stamp is automatically set to the current date and time.

Examples

Adding a Column to a Table

The following ALTER TABLE statement adds a new column, STATUS, to the CONSULTANT table. The value of STATUS in all existing rows is blank because the statement specifies WITH DEFAULT.

alter table consultant
    add status character(1) not null with default;

Further Restricting Data in a Table

The following ALTER TABLE statement defines an additional restriction on the data that can be stored in the CONSULTANT table. CA IDMS adds the constraint only if each existing row of the CONSULTANT table already has 'A' or 'I' in the STATUS column.

alter table consultant
    add check (status in ('A', 'I'));

Change a column's data type:

alter table demo.empl
    alter column city
    set data type varchar(20);

Drop a column using the CASCADE option:

alter table demo.empl
    drop column status cascade;

Rename a column:

alter table demo.empl
    rename column proj_id to project_id;

Adding a default index:

alter table emp.dept
    add default index;
More Information

- For more information about defining tables, see CREATE TABLE and DROP TABLE.
- For more information about implementing indexes, see the CA IDMS Database Design Guide.
- For more information about displaying a table, see DISPLAY/PUNCH TABLE.

**ALTER TABLE PROCEDURE**

The ALTER TABLE PROCEDURE data description statement modifies the definition of a table procedure in the dictionary. It is also a CA IDMS extension of the SQL standard. Using the ALTER TABLE PROCEDURE statement, you can:

- Add a new parameter to a table procedure
- Revise the estimated row and I/O counts
- Change the external name of the procedure
- Change the size and characteristics of the work areas passed to the procedure
- Change the execution mode of the procedure
- Update the table procedure's synchronization timestamp
- Change the table procedure's default database option
- Change the table procedure's transaction sharing option

**Authorization**

To issue an ALTER TABLE PROCEDURE statement, you must own or hold the ALTER privilege on the table procedure named in the statement.

**Syntax**

```
ALTER TABLE PROCEDURE [schema-name.] table-procedure-identifier
```

- **ADD parameter-definition**
- **EXTERNAL NAME external-routine-name**
- **ESTIMATED ROWS row-count**
- **ESTIMATED I/OS io-count**
- **LOCAL WORK AREA local-stge-size**
- **GLOBAL WORK AREA global-stge-size**
- **KEY key-ID [key-ID] [NULL]**
- **USER MODE**
- **SYSTEM MODE**
- **TIMESTAMP timestamp-value**
- **DEFAULT DATABASE [NULL CURRENT] [CURRENT]**
- **ON [OFF DEFAULT]**
Expansion of parameter-definition

\[ \text{parameter-name} \quad \text{data-type} \quad \text{WITH DEFAULT} \]

Parameters

**table-procedure-identifier**

Specifies the name of the table procedure being modified.  
*Table-procedure-identifier* must identify a table procedure defined in the dictionary.

**schema-name**

Identifies the schema associated with the named table procedure.  If you do not specify a *schema-name*, it defaults to:

- The current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically
- The schema associated with the access module used at runtime, if the statement is embedded in an application program

**parameter-definition**

Defines one or more new parameters to be associated with the table procedure.  
New parameters are added, in the order specified, after the last existing parameter.

For a description of *parameter-definition*, see CREATE TABLE PROCEDURE.

**external-routine-name**

 Specifies the one- to eight-character name of the program which CA IDMS calls to process references to the table procedure.
**row-count**

Specifies an integer value, in the range of 0 through 2,147,483,647, which represents the average number of rows that the table procedure returns for a given set of input parameters.

**io-count**

Specifies an integer value, in the range of 0 through 2,147,483,647, which represents the average number of disk accesses that the table procedure generates for a given set of input parameters.

**local-stge-size**

Specifies an integer, in the range of 0 through 32767, which represents the size, in bytes, of a local storage area that CA IDMS allocates at runtime and passes to the table procedure on each invocation.

CA IDMS allocates a local storage area on the first call to a table procedure for each SQL statement within a transaction or for a set of SQL statements related through reference to the same cursor. OPEN, CLOSE, FETCH, POSITIONED UPDATE and POSITIONED DELETE statements are related through a cursor. CA IDMS passes the same local storage area to the table procedure for all calls for one statement or related statements. CA IDMS releases the local work area when the SQL statement has completed execution or at the time the cursor is closed.

**global-stge-size**

Specifies an integer, in the range of 0 through 32767, which represents the size, in bytes, of a global storage area that CA IDMS allocates at runtime and passes to the table procedure on each invocation.

CA IDMS allocates a global storage area once within a transaction and retains it until the transaction terminates.

**key-id**

Specifies the one- to four-character identifier for the global storage area. CA IDMS passes the same piece of global storage within a transaction to all routines that have the same global storage key.

If you do not specify a storage key, its value remains unchanged. To remove a storage key, specify **NULL** as the key.
USER MODE

Specifies the table procedure should execute as a user-mode application program within CA IDMS.

SYSTEM MODE

Specifies the table procedure should execute as a system-mode application program. To execute as a system mode application, the program must be fully reentrant and be written in either:

- Assembler using DC calling conventions
- COBOL or PL/I and compiled with an LE-compliant compiler

**timestamp-value**

Specifies the value of the synchronization stamp to be assigned to the table procedure. *Timestamp-value* must be a valid external representation of a timestamp.

DEFAULT DATABASE

Specifies whether a default database should be established for database sessions started by the table procedure.

**NULL**

Specifies that no default database should be established.

**CURRENT**

Specifies that the database to which the SQL session is connected should become the default for any database session started by the table procedure.

TRANSACTION SHARING

Specifies whether transaction sharing should be enabled for database sessions started by the table procedure. If transaction sharing is enabled for a table procedure's database session, it will share the current SQL session's transaction.

**ON**

Specifies that transaction sharing should be enabled.

**OFF**

Specifies that transaction sharing should be disabled.

**DEFAULT**

Specifies that the transaction sharing setting that is in effect when the procedure is invoked should be retained.
Parameters for Expansion of parameter-definition

**parameter-name**

Specifies a 1- to 32-character name of a parameter to be passed to the table procedure. *Parameter-name* must:

- Be unique within the table procedure that you are defining
- Follow the conventions for SQL identifiers

All parameters are implicitly nullable. Input parameters can be assigned NULL as a parameter value and output parameters can return NULL.

**data-type**

Defines the data type for the named parameter. For expanded data-type syntax, see Expansion of Data-type.

**WITH DEFAULT**

Directs CA IDMS to pass a default value for the named parameter if no value for the parameter is specified.

The default value for a parameter is based on its data type:

<table>
<thead>
<tr>
<th>Column data type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>Blanks</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>A character string literal with a length of zero (that is, '')</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>Double-byte blanks</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>A double-byte character string literal with a length of zero</td>
</tr>
<tr>
<td>DATE</td>
<td>The value in the CURRENT DATE special register</td>
</tr>
<tr>
<td>TIME</td>
<td>The value in the CURRENT TIME special register</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>The value in the CURRENT TIMESTAMP special register</td>
</tr>
<tr>
<td>All numeric data types</td>
<td>0 (zero)</td>
</tr>
</tbody>
</table>

**Usage**

**Specifying a Synchronization Stamp**

When defining or altering a table procedure, you can specify a value for its synchronization stamp. You should use care when doing so because the purpose of the stamp is to enable the detection of discrepancies between an entity and its definition. If explicitly specified, you must set the synchronization stamp to a new value following a change so that the change is detectable by the runtime system.

If not specified, the synchronization stamp is automatically set to the current date and time.
Examples

Adding Parameters to a Table Procedure

The following ALTER TABLE PROCEDURE statement adds two new parameters to the EMP.ORG table procedure:

```
alter table procedure emp.org
    add (job_level    decimal(1),
         job_title    char(20));
```

BEGIN DECLARE SECTION

The BEGIN DECLARE SECTION statement notifies the precompiler that an SQL declare section follows. You can use this statement only in SQL that is embedded in a program.

Authorization

None required.

Syntax

```
BEGIN
DECLARE SECTION
```

Parameter

```
BEGIN DECLARE SECTION.
```

Notifies the precompiler that an SQL declare section follows. An SQL declare section contains the definition of one or more host variables.

Usage

Declaring Host Variables

You can:

- Place an SQL declare section in an application program wherever host language rules allow variable declarations
- Define any number of host variable declarations in an SQL declare section
- Include any number of SQL declare sections in a single application program
Example

Beginning and Ending an SQL Declaration Section

In this COBOL example, BEGIN DECLARE SECTION begins an SQL declare section and END DECLARE SECTION ends it. The SQL declare section contains the definition of five host variables.

```
WORKING-STORAGE SECTION.
  .
  .
EXEC SQL BEGIN DECLARE SECTION END-EXEC
  01 HV-EMP-ID   PIC S9(8) USAGE COMP.
  01 HV-EMP-LNAME PIC X(20).
  01 HV-SALARY-AMOUNT PIC S9(6)V(2) USAGE COMP-3.
  01 HV-PROMO-DATE PIC X(10).
  01 HV-PROMO-DATE-I PIC S9(4) USAGE COMP.
EXEC SQL END DECLARE SECTION END-EXEC
```

More Information

- For more information about ending an SQL declare section, see END DECLARE SECTION.
- For more information about declaring host variables, see Host Variables or the CA IDMS SQL Programming Guide.

CALL

The CALL data manipulation statement executes a procedure or a table procedure. The values of output parameters return in the form of 0 to 1 result row for the call of a procedure and 0 to multiple rows for the call of a table procedure. When the CALL statement is:

- Submitted through the command facility, the values of all parameters are contained in the result rows and displayed in tabular form.
- Embedded in an application program, at most a single row can return and the values in the result row are stored in host variables.
- Dynamically prepared, the result rows must return through a cursor just as if the prepared statement were a SELECT statement.

Authorization

To issue a CALL statement, you must either own or have the SELECT privilege on the procedure or table procedure explicitly named in the statement.
CALL

Chapter 8: Statements  307

Syntax

CALL procedure-reference table-procedure-reference

Parameters

procedure-reference

Identifies the procedure that is invoked, the input values that pass to the procedure and optionally the host.variables for passing and returning values of input/output parameters.

table-procedure-reference

Identifies the table procedure that is invoked, the input values that pass to the table procedure and optionally the host.variables for passing and returning values of input/output parameters.

Usage

Embedding a CALL statement

When embedding a CALL statement in an application program, output values return only for those parameters that you specify as host-variables, local variables, or routine parameters.

call myproc (5, :wk-out)

If the procedure or table procedure updates the value of the first parameter and the application program needs to see that value, then you must specify both parameters as host-variables. You should set the first host-variable to 5 before executing the CALL statement:

move 5 to wk-val
call myproc (:wk-val, :wk-out)

Initializing parameters

It is important to initialize all host-variables, local variables, and routine parameters that you reference in a CALL statement prior to its execution. Since all such parameters are treated as input values, failure to initialize such host-variables, local variables, or routine parameters results in a data exception if its value does not conform to its data type. If there is no value to pass, you should declare the host-variable with a null indicator with a value set to -1 and set the local variables and routine parameters to null.
Dynamically executing a CALL statement

When describing the output from a dynamically prepared CALL statement, the SQLD field of the SQLDA contains a count of the number of parameters for the procedure or table procedure. The first SQLD entries within the SQLDA contain a description of those parameters.

You must return the output parameter values of a dynamically prepared CALL statement using a cursor. In other words, you must treat a dynamically prepared CALL statement as a dynamically prepared SELECT statement.

Result Sets from SQL-invoked Procedures

An SQL-invoked procedure can return results to the caller by assigning values to one or more parameters of the procedure. Using Dynamic Result Sets, an SQL-invoked procedure can return result sets in the form of rows of result tables.

To exploit result sets returned by an SQL-invoked procedure, an application must consist of at least an SQL-invoked procedure and a caller of that procedure. The caller can be an SQL client program or another SQL-invoked routine. The SQL-invoked procedure that returns the result sets can be an external procedure (COBOL, PL/I, Assembler or CA ADS) or an internal SQL procedure written in SQL.

For an SQL-invoked procedure to return result sets to its caller, it must be defined with a positive integer value for the new Dynamic Result Sets attribute.

A cursor declared or dynamically allocated in the SQL-invoked procedure becomes a potential returned result set if its definition contains With Return as the value for the new returnability attribute. Such a cursor is called a returnable cursor. It becomes a returned result set if it is in the open state when the SQL-invoked procedure terminates.

An SQL-invoked procedure can return multiple result sets up to the number specified by the Dynamic Result Sets attribute of the procedure. The list of returned result sets are sequenced in the order of the open of the cursors. If the procedure starts multiple sessions, then returned result sets are grouped by session and the sessions are sequenced in the order of the connects. After a procedure CALL, the new SQLCA field SQLCNRRS contains the number of result sets returned by the procedure.
The caller of an SQL-invoked procedure accesses returned result sets by allocating a dynamic cursor and associating it with the procedure through an `ALLOCATE CURSOR FOR PROCEDURE` statement. Such a cursor is called a **received cursor**.

A successful `ALLOCATE CURSOR FOR PROCEDURE` statement associates the received cursor with the first result set from the sequence of returned result sets and places the cursor in the open state. The cursor position is the same as it was when the SQL-invoked procedure terminated and the associated returned result set is removed from the list of returned result sets.

The caller of the procedure can access the next in the sequence of returned result sets by either allocating another cursor for the procedure or by closing the previously allocated received cursor. If the close is successful and the list of remaining returned result sets is not empty, the received cursor is automatically placed in the open state and associated with the result set that is now first in the list. The newly associated result set is also removed from the list. This process can be repeated until the list of returned result sets is empty.

A new invocation of the SQL-invoked procedure automatically destroys all the returned result sets from the previous invocation.

The received cursors, allocated by the caller and associated with returned result sets, are necessarily dynamic. Unless the program knows the returned columns and their data type, a `DESCRIBE CURSOR` statement is needed to retrieve the description of the returned result set in an SQL descriptor area (SQLDA).

Only the immediate caller of an SQL-invoked procedure can process returned result sets. There is no mechanism for the caller to return returned result sets to its caller.
Calling an SQL Procedure

An SQL procedure is an SQL-invoked procedure with language SQL. Any transaction started by this procedure is shared with the transaction of the caller. After returning from an SQL procedure, any session opened by the procedure is automatically released except for sessions that have result sets. Such sessions are released when their last result set has been processed and the associated received cursor has been closed.

Note: When calling an SQL procedure or table procedure, if error message DB001078 with condition 38999 is returned, it might indicate that a record associated with the dialog was too large to fit into the buffer. If this occurs, see messages DC171027 and DC466014 in the IDMS log for more information on the dialog and process causing this problem and how to resolve it.

Calling an SQL-invoked Procedure Returning Result Sets

After a CALL of an SQL-invoked procedure that has been defined with a positive value for the Dynamic Result Sets attribute the number of actual returned result sets is available in the field SQLCNRRS of the SQLCA. The number of returned result sets can also be determined by issuing a GET DIAGNOSTICS statement to retrieve the IDMS_RETURNED_RESULT_SETS information item.

The successful execution of a CALL statement may result in one of two warning conditions:

0100C SQL invoked procedure returned result sets

Indicates that the number of result sets returned by the procedure is less than or equal to the value of the procedure's DYNAMIC RESULT SETS attribute.

0100E Attempt to return too many result sets

Indicates that the procedure attempted to return more result sets than permitted by its DYNAMIC RESULT SETS attribute. The actual number of result sets is reduced to the value of the DYNAMIC RESULT SETS attribute.

A call of a procedure destroys any result sets left over from a previous invocation of the same procedure.

Example

The following example illustrates the basic coding techniques to use dynamic result sets in an application. The SQL procedure, SQLROUT.PROCESSRESULTSET, calls the SQL procedure SQLROUT.CREATERESULTSET and dynamically processes the returned result sets. Included in the example are the definition of a table SQLROUT.RSTAB, the load of this table, the definitions of the SQL procedures SQLROUT.CREATERESULTSET and SQLROUT.PROCESSRESULTSET, and finally the CALL of SQLROUT.PROCESSRESULTSET.
create table SQLROUT.RSTAB
  ( ID integer,
    MES character(10)
  ) in PROJSEG.PROJAREA ++
insert into SQLROUT.RSTAB values (1, 'txt1')++
insert into SQLROUT.RSTAB values (2, 'txt2')++
insert into SQLROUT.RSTAB values (3, 'txt3')++
insert into SQLROUT.RSTAB values (4, 'txt4')++
insert into SQLROUT.RSTAB values (5, 'txt5')++
insert into SQLROUT.RSTAB values (6, 'txt6')++
commit++

create procedure SQLROUT.CREATERESULTSET
  (TITLE char(10) with default,
   P_ID integer with default,
   RESULT char(30) with default
  )
  external name CRRESSET language SQL
dynamic result sets 4
begin not atomic
  declare DYNST char(100);
  declare L_ID integer default 2;
  declare TEST_CUR2 cursor with return for
    select ID, MES from SQLROUT.RSTAB
    where ID >= P_ID;
  declare TEST_CUR4 cursor with return for
    select ID, MES from SQLROUT.RSTAB
    where ID < P_ID;
  set DYNST = 'SELECT ID, MES FROM SQLROUT.RSTAB '|| 'WHERE ID < CAST(? AS INTEGER)';
  prepare 'DYNSTMT1' FROM DYNST;
  allocate 'TEST_CUR1' cursor with return for 'DYNSTMT1';
  prepare 'DYNSTMT3' FROM DYNST;
  allocate 'TEST_CUR3' cursor with return for 'DYNSTMT3';
  open TEST_CUR4;
  open 'TEST_CUR3' using L_ID;
  set L_ID = L_ID + 1;
  open 'TEST_CUR1' using L_ID;
  open TEST_CUR2;
  set RESULT = '4 RESULT SET RETURNED';
end++
commit++

create procedure SQLROUT.PROCESSRESULTSET
  ( TITLE character(10) with default,
    P_ID integer with default,
    CNT_RESULT_SETS integer,
    RESULT varchar(1024) with default,
begin not atomic
    declare L_MES char(20);
    declare L_ID integer;
    declare BINBUF binary(200);
    declare CNT integer;
    declare L_CNT_RESULT_SETS integer default 0;
    declare continue handler for SQLWARNING
        set RESULT = RESULT|| SQLSTATE|| ' ';
    declare exit handler for SQLEXCEPTION
        begin
            declare C_FUN CHAR(64);
            declare L_MES varchar(256);
            declare M_TEXT CHAR(256);
            get diagnostics C_FUN = COMMAND_FUNCTION;
            get diagnostics CONDITION 1 M_TEXT = MESSAGE_TEXT;
            set ERROR1 = TRIM(CHAR(CNT))|| ' ROWS FETCHED; '|| TRIM(C_FUN)|| ' '|| TRIM(M_TEXT);
            get DIAGNOSTICS CONDITION 2 M_TEXT = MESSAGE_TEXT;
            set ERROR1 = ERROR1|| TRIM(M_TEXT);
        end;
    set RESULT = 'ROWS FETCHED: ';
    call SQLROUT.CREATERESULTSET(TITLE,P_ID, L_MES);
    get diagnostics CNT_RESULT_SETS = IDMS_RETURNED_RESULT_SETS;
    allocate 'CUR2' for procedure
        specific procedure SQLROUT.CREATERESULTSET;
    while (L_CNT_RESULT_SETS < CNT_RESULT_SETS) do
        set CNT = 0;
        describe cursor 'CUR2' structure using SQL descriptor SQLDA;
        fetch 'CUR2' into L_ID, L_MES;
        while (SQLSTATE = '00000') do
            set RESULT = RESULT|| '<'|| trim(L_MES)|| '>'; 
            set CNT = CNT + 1;
            fetch 'CUR2' into L_ID, L_MES;
        end while;
        close 'CUR2';
        set L_CNT_RESULT_SETS = L_CNT_RESULT_SETS + 1;
        set RESULT = RESULT|| '# '|| trim(char(CNT))|| '/';
    end while;
end
++
commit++
CLOSE

The CLOSE statement places a specified cursor in the closed state or disassociates a received cursor from the current returned result set and associates it with the next result set returned by the procedure. Use this statement only in SQL that is embedded in a program.

Authorization

None required.

Syntax

```
CLOSE cursor-name
```
**Parameter**

cursor-name

Specifies the cursor to be closed. **Cursor-name** must identify a cursor previously defined by a DECLARE CURSOR statement within the application program or by an ALLOCATE CURSOR statement executed within the same SQL transaction.

**Usage**

Automatic Closing of Cursors

The COMMIT statement without the CONTINUE clause and the ROLLBACK statement automatically close all open cursors used by the application program.

Closing Declared Global Cursors

Cursors declared with the GLOBAL option may be closed only within the same application program in which the global cursor declaration was made. This restriction does not apply to global cursors defined using an ALLOCATE CURSOR statement.

Closing a Received Cursor

A received cursor is a dynamically allocated cursor used to process one or more result sets returned by an SQL-invoked procedure. Returned result sets are maintained in an ordered list. An ALLOCATE CURSOR statement associates the cursor with the first result set in the list and removes it from the list.

If the list of returned result sets is not empty when a received cursor is closed, the CLOSE statement causes the following actions to be taken:

- Disassociates the cursor from its current result set
- Associates the cursor with the first result set in the list of returned result sets
- Removes the result set from the list
- Positions the cursor at the same point at which the corresponding returnable cursor was left by the procedure
- Returns a warning "Additional result sets returned" (SQLSTATE "0100D)

Closing the cursor associated with the last result set of a session started by the called procedure, releases that session.
Examples

Closing a Cursor

The following CLOSE statement places the cursor named ALL_EMP_CURSOR in the closed state:

EXEC SQL
  CLOSE ALL_EMP_CURSOR
END-EXEC

Closing a Global Dynamic Cursor

The following statement closes the global cursor whose name is passed in :CNAME:

EXEC SQL
  CLOSE GLOBAL :CNAME
END-EXEC

More Information

- For more information about defining and manipulating cursors, see DECLARE CURSOR, FETCH, and OPEN.
- For more information about using cursors in an application program, see the CA IDMS SQL Programming Guide.
- For more information about closing a received cursor, see ALLOCATE CURSOR.

COMMIT

The COMMIT transaction management statement requests that changes to the database made by the SQL session be made permanent. Optionally, the SQL session continues or terminates.

Authorization

None required.

Syntax

```
COMMIT work
```
Parameters

**CONTINUE**

Directs CA IDMS to maintain the state of the SQL session after committing changes to the database. This has the effect of maintaining the position of open cursors, retaining temporary tables and associating a new transaction with the session after terminating the current one.

If you do not specify CONTINUE, CA IDMS terminates the current transaction and does not start a new one. It also closes all open cursors and drops all temporary tables. The CONTINUE parameter is a CA IDMS extension of the SQL standard.

**RELEASE**

Directs CA IDMS to end the current SQL session as well as the current transaction after committing the changes to the database. The RELEASE parameter is a CA IDMS extension of the SQL standard.

Usage

**Effect of a COMMIT on the SQL Session's Transaction**

If the SQL session's database transaction is not shared, a COMMIT statement has the following impact on its transaction:

- Commits all changes made by the SQL session
- Releases all exclusive locks
- Terminates the transaction
- Associates a new transaction with the SQL session if CONTINUE is specified

**Effect of a COMMIT on an SQL Session**

A COMMIT statement without the CONTINUE option has the following impact on the SQL session:

- Releases all locks used to protect cursor currencies
- Closes all open cursors
- Drops all temporary tables
- Deletes all dynamically compiled statements
- Replaces the access module in the dictionary if the module was recreated during transaction execution
- Terminates the SQL session if CA IDMS connected it automatically or if RELEASE is specified

If CONTINUE is specified, the COMMIT statement impacts the SQL session's transaction but has no impact on the session itself. Its state remains as it was before the COMMIT statement was issued.
Effect of Transaction Sharing

A COMMIT statement requests that changes made by an SQL session be committed. However, if more than one database session is sharing the session's transaction, those changes might not be committed immediately. All sharing sessions that have had activity since the last commit, rollback or session start must signal their willingness to commit by issuing a COMMIT statement before changes are actually made permanent. The last one to do so causes the transaction to be committed.

A teleprocessing commit statement such as a COMMIT TASK can be used to cause the immediate committing of a shared transaction, since it impacts all of the associated sessions. A COMMIT issued by an encompassing session automatically commits all of its subordinate sessions.

Example

Committing Changes to the Database

The following COMMIT statement commits changes made during the current transaction to the database, but does not end the transaction:

```
EXEC SQL
  COMMIT CONTINUE
END-EXEC
```

More Information

- For more information about ending a transaction without committing changes to the database, see ROLLBACK.
- For more information about ending an SQL session, see RELEASE.
- For more information about managing transactions, see the *CA IDMS SQL Programming Guide*.
- For more information about establishing an SQL session, see CONNECT.

CONNECT

The CONNECT session management statement begins an SQL session by connecting to a CA IDMS dictionary. The dictionary you specify must contain the definitions of the database to be accessed during the session.
Authorization

To issue a CONNECT statement:

- In central version, you must have the authority to sign on to the DC/UCF system with which the dictionary is associated.
- In local mode, no privileges are required.

Syntax

```
CONNECT TO dictionary-name
```

Parameters

**dictionary-name**

Specifies the name of the dictionary to which the session is connected.

**:dictionary-variable-name**

Identifies a host variable containing the name of the dictionary to which the session is connected. *Dictionary-variable-name* must be a host variable previously declared in the application program.

**dictionary-sqlvariable-name**

Identifies a routine parameter or local variable containing the name of the dictionary to which the session will be connected. *Dictionary-sqlvariable-name* must be previously declared in the SQL routine.

You can specify **:dictionary-variable-name or dictionary-sqlvariable-name** only when you embed the CONNECT statement in an application program or SQL routine.
Usage

Ending an SQL Session

If you use the CONNECT statement to begin an SQL session, you must end the session with one of the following statements:

- RELEASE
- COMMIT RELEASE
- ROLLBACK RELEASE

Automatic Connection

The CONNECT statement is not required to establish a connection. CA IDMS automatically attempts to establish a connection upon executing the first SQL statement. When establishing an automatic connection, CA IDMS establishes a connection to a default dictionary.

Specifying a Dictionary Name

The name specified on a CONNECT statement should be associated with a DDLDML area, a DDLCAT area; or a DDLDML and a DDLCAT area. The name can represent:

- A DBNAME that contains the appropriate segments
- An individual segment

If you specify a DBNAME, it can include segments in addition to those for the dictionary itself.

Examples

Specifying the Dictionary Name

The following CONNECT statement establishes a connection to the dictionary named EMPDICT.

connect to empdict;

Using a Host Variable

The following CONNECT statement establishes a connection to the dictionary name contained in the host variable :DICT-NAME:

EXEC SQL
   CONNECT TO :DICT-NAME
END-EXEC
More Information

- For more information about releasing a connection and ending an SQL session, see `RELEASE`.
- For more information about user profiles, see the *CA IDMS Security Administration Guide*.
- For more information about system profiles and DCUF SET DICTNAME, see the *CA IDMS System Tasks and Operator Commands Guide*.
- For more information about host variables, see Host Variables.
- For more information about managing SQL sessions, see the *CA IDMS SQL Programming Guide*.
- For more information about how the default dictionary is determined, see the *CA IDMS SQL Programming Guide*.
- For more information about the components of a dictionary, see the *CA IDMS Database Administration Guide*.

CREATE ACCESS MODULE

The CREATE ACCESS MODULE statement creates an access module from one or more RCMs. CA IDMS stores the access module definition and the access module itself in the dictionary. It is also a CA IDMS extension of the SQL standard.

Authorization

To issue a CREATE ACCESS MODULE statement, you must own the schema with which the access module is being associated or hold the CREATE privilege on the named access module.

In addition to enforcing this authorization requirement, CA IDMS also validates the access module owner's authority to execute every DML statement in the RCMs included in the access module if the dictionary to which the SQL session is connected is controlled by CA IDMS internal security.

If the access module owner does not hold the authority to execute a DML statement in the access module, when the access module is created, a warning is issued. If the owner still lacks a necessary authority when the access module is executed, an error is returned.
Syntax

```
CREATE ACCESS MODULE schema-name. access-module-name

VERSION am-version-number

FROM dictionary-name. rcm-name VERSION rcm-version-number

MAP schema-name-1 TO schema-name-2

AUTO RECREATE ON OFF

VALIDATE BY STATEMENT BY MODULE ALL

READ ONLY READ WRITE

DEFAULT ISOLATION CURSOR STABILITY TRANSIENT READ

READY segment-name.area-name ready-options

Expansion of ready-options

- SHARED RETRIEVAL
- SHARED UPDATE
- PROTECTED RETRIEVAL
- PROTECTED UPDATE
- EXCLUSIVE

- INCREMENTAL
- PRECLAIM
```
Parameters

**access-module-name**

Specifies the name of the access module being created. *Access-module-name* must be a one- through eight-character name that follows the conventions for SQL identifiers.

The combination of *access-module-name* and *am-version-number* must be unique within the dictionary. Multiple access modules with the same *access-module-name* can be associated with a given schema provided they have different version numbers.

**schema-name**

Specifies the schema to be associated with the access module. *Schema-name* must identify a schema defined in the dictionary.

The owner of the schema with which the access module is associated implicitly becomes owner of the access module.

If you do not specify *schema-name*, CA IDMS uses the current schema in effect for your SQL session.

**am-version-number**

Specifies the version number of the access module to be created.

If the specified version of the access module already exists, an error is returned.

If you do not specify VERSION, *am-version-number* is set to 1.

**FROM rcm-name**

Specifies one or more RCMs from which CA IDMS is to create the access module. *Rcm-name* must identify an RCM stored in the dictionary and must be unique within the list of RCM names.

**dictionary-name**

Identifies the dictionary in which the named RCM resides.

If you do not specify *dictionary-name*, it is set to the name of the dictionary to which the SQL session is connected.

**rcm-version-number**

Specifies the version of the RCM to be included in the access module.

If you do not specify *rcm-version-number*:

1. CA IDMS looks for an RCM with a version number that matches *am-version-number*
2. If no such RCM is found, CA IDMS looks for version 1
3. If CA IDMS does not find a match, it issues a warning
MAP

Specifies one or more mappings for schema names that qualify table and view identifiers in data manipulation statements.

If you do not specify MAP, table and view identifiers are not replaced. If a table or view has no qualifier, CA IDMS uses the schema name of the access module as the qualifier.

**schema-name-1**

Directs CA IDMS to replace occurrences of the specified schema name with the schema name specified in the TO parameter.

**NULL**

Directs CA IDMS to use the schema name specified in the TO parameter as the qualifier for unqualified table and view identifiers.

**TO schema-name-2**

Directs CA IDMS to use the specified schema name as the replacement for **schema-name-1** or as the qualifier for unqualified table and view identifiers.

**AUTO RECREATE**

Specifies whether CA IDMS is to re-create the access module after detecting any of the following:

- An attempt to execute an uncompiled statement
- A change to the definition of a table referenced in the access module
- The execution of a program that has been recompiled since its RCM was included in the access module

CA IDMS identifies the above conditions by comparing definition timestamps in the access module to corresponding timestamps in the database and the host program.

If you do not specify AUTO RECREATE, the default is ON.

**ON**

Directs CA IDMS to re-create the access module at runtime when timestamps do not match. CA IDMS continues the current transaction with the re-created access module but does not replace the access module in the dictionary until the transaction terminates with a COMMIT statement.

**OFF**

Directs CA IDMS not to re-create the access module at runtime. If CA IDMS detects a mismatch in timestamps, it returns an error and terminates the current transaction.
VALIDATE

Indicates when CA IDMS is to check the definition timestamps of tables in the access module to ensure that the definition has not changed since the access module was created or last altered.

If you do not specify VALIDATE, the default is VALIDATE ALL.

BY STATEMENT

Directs CA IDMS to check the definition timestamp for a table immediately before executing the first statement in the access module that references the table.

BY MODULE

Directs CA IDMS to check the definition timestamp for each table referenced by a statement in an RCM immediately before executing the first statement in the RCM.

ALL

Directs CA IDMS to check the definition timestamp for each table in the access module immediately before executing the first statement in the access module.

READ ONLY

Specifies transactions started by the access module that do not execute a SET TRANSACTION statement can retrieve data but cannot update the database.

READ WRITE

Specifies transactions started by the access module that do not execute a SET TRANSACTION statement can retrieve data and update the database.
DEFAULT ISOLATION

Specifies the isolation level of transactions started by the access module that do not execute a SET TRANSACTION statement.

At runtime, the isolation level of a transaction determines the length of time retrieval locks are held for the purpose of insulating the transaction from the effects of other concurrent transactions. (Update locks are always held until the transaction is committed or rolled back.)

If you do not specify DEFAULT ISOLATION, the default is CURSOR STABILITY.

CURSOR STABILITY

Specifies the default isolation level for a transaction is cursor stability.

An isolation level of cursor stability guarantees read integrity. Read integrity ensures that:

■ All data read by the transaction is in a committed state
■ The current row of an updateable cursor is protected from update by other transactions while it remains current

TRANSIENT READ

Specifies the default isolation level for a transaction is transient read.

An isolation level of transient read provides no guarantees of read integrity. A transaction executing under transient read cannot perform updates to the database. CA IDMS does not maintain any locks for a transaction with an isolation level of transient read.

The combination of TRANSIENT READ and a transaction state of READ WRITE is invalid. Thus, if you specify TRANSIENT READ, CA IDMS assumes a transaction state of READ ONLY.

READY

Specifies a ready mode for one or more areas accessed through the access module, and specifies when the ready occurs.

The ready mode associated with an area determines:

■ Under the central version, the ready mode in which transactions access the area. The ready mode determines the types of area and row locks CA IDMS places for a transaction.
■ In local mode, the type of physical lock CA IDMS places on the area.

If you do not specify READY, the ready options for all areas used by the access module are determined at runtime by:

■ The transaction state (READ WRITE or READ ONLY)
■ The isolation level
■ The availability of the area under the central version

Note: For more information, see "Usage," following these parameter descriptions.
**segment-name**

Identifies the segment associated with the area to which the following ready options apply.

If the access module is used to access a non-SQL-defined database, `segment-name` is optional. In this case, if you do not specify `segment-name`, CA IDMS accesses the first segment for which it finds a match on `area-name`.

**area-name**

Specifies the name of the area to which the following ready options apply. `Area-name` must identify an area used by the access module.

**ALL**

Specifies the following ready options apply to all areas in the access module.

**Parameters for Expansion of ready-options**

The ready-options are used for a specified area or for all areas in the access module. Expanded syntax for `ready-options` is shown immediately following the CREATE ACCESS MODULE syntax.
SHARE RETRIEVAL

Specifies a transaction can retrieve, but not update, data in the area. Other concurrent transactions can retrieve and update data in the area.

SHARE UPDATE

Specifies that:

■ Under the central version, transactions access the indicated areas in shared update mode.

With access to an area in shared update mode, a transaction can retrieve and update data in the area. Other concurrent central version transactions can also both retrieve and update data in the area.

■ In local mode, CA IDMS first places a physical lock on the indicated areas.

With a physical lock on an area, a local mode transaction can retrieve and update data in the area. Concurrent transactions executing in other address spaces can retrieve but not update data in the area.

PROTECTED RETRIEVAL

Specifies that:

■ Under the central version, transactions access the indicated areas in protected retrieval mode.

With access to an area in protected retrieval mode, a transaction can retrieve, but not update, data in the area. Other concurrent central version transactions can also retrieve, but not update, data in the area.

■ In local mode, a ready mode of PROTECTED RETRIEVAL is equivalent to a ready mode of SHARED RETRIEVAL.
PROTECTED UPDATE

Specifies that:

■ Under the central version, transactions access the indicated areas in protected update mode. With access to an area in protected update mode, a transaction can retrieve and update data in the area. Other concurrent central version transactions can retrieve, but not update, data in the area.

■ In local mode, a ready mode of PROTECTED UPDATE is equivalent to a ready mode of SHARED UPDATE.

EXCLUSIVE

Specifies that:

■ Under the central version, transactions access the indicated areas in exclusive mode. With access to an area in exclusive mode, a transaction can retrieve and update data in the area. All other concurrent central version transactions can neither retrieve nor update data in the area except transactions with an isolation level of transient read, which can retrieve data in the area.

■ In local mode, a ready mode of EXCLUSIVE is equivalent to a ready mode of SHARED UPDATE.

INCREMENTAL

Directs CA IDMS to defer the ready of each indicated area until execution of the first statement in the access module that requires access to the area.

PRECLAIM

Directs CA IDMS to ready each indicated area when executing the first statement in the access module that requires database or dictionary access.
Usage

Automatic Access Module Recreation

An automatic recreation of the access module occurs when CA IDMS detects a change in the definition of a table referenced in the access module.

The scope of what is recreated is limited by how you specify the VALIDATE option, as described in the following table:

<table>
<thead>
<tr>
<th>If validation is by</th>
<th>CA IDMS recompiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATEMENT</td>
<td>Only the statement just checked; other statements which reference the same table or another table with a changed definition are recompiled as they are encountered</td>
</tr>
<tr>
<td>MODULE</td>
<td>All statements in the current RCM that reference tables with changed definitions when the first such statement is encountered</td>
</tr>
<tr>
<td>ALL</td>
<td>All statements in the access module that reference tables with changed definitions when the first such statement is encountered</td>
</tr>
</tbody>
</table>

Repeatability of Retrieval Operations

An isolation level of cursor stability assures that data currently being accessed by a transaction is protected from update by other transactions. Cursor stability does not protect data that was accessed previously by the transaction.

Therefore, a cursor might return six rows the first time it is opened and five rows the second time, even though both operations are performed within the same transaction and that transaction has not made intervening updates. The discrepancy would be caused by updates by other transactions executing concurrently.

To completely isolate a transaction from the effects of other transactions, specify a protected ready mode for the areas that the transaction accesses. A ready mode of protected retrieval for retrieval applications and protected update for update applications ensures the repeatability of retrieval operations.
Runtime Ready Modes

The ready mode in which an area is accessed at runtime depends on the requested ready mode, the transaction state, the isolation level, and the area's availability:

- If the transaction state and isolation level are READ ONLY and TRANSIENT READ, all areas are accessed using transient retrieval mode, in which no row locks are placed.
- If the transaction state and isolation level are READ ONLY and CURSOR STABILITY, all areas are accessed using retrieval modes only.

  If update modes were specified on the CREATE or ALTER ACCESS MODULE statement, they are changed to shared retrieval, and if no ready option was specified, the default is shared retrieval.

- If the transaction state and isolation level are READ WRITE and CURSOR STABILITY, all areas are accessed using the mode specified on the CREATE ACCESS MODULE.

  If no mode was specified, the default is:
  - Shared update under the central version if the area status is update
  - Shared update in local and no other copy of IDMS has update control of the area
  - Shared retrieval under the central version if the area status is retrieval
  - Shared update in local if another copy of IDMS has update control of the area

Under central version, if an area is being readied in a retrieval mode and the status of the area is transient retrieval, the ready mode is changed to transient retrieval.

Ready Modes and Area Status Under the Central Version

The ready mode in which a central version transaction obtains access to an area must be compatible with the status of the area within the DC/UCF system. If the area’s status is:

- Update, transactions executing under the system can obtain access to the area in any ready mode
- Retrieval or transient retrieval, transactions executing under the system can obtain access to the area in a retrieval ready mode only
- Offline to the system, transactions executing under the system cannot obtain access to the area
Shared, Protected, and Exclusive Ready Modes

In the shared ready modes (shared retrieval and shared update), CA IDMS provides protection from the effects of other transactions at the row level. In the protected ready modes (protected retrieval and protected update), CA IDMS provides protection at the area level. The shared ready modes, therefore, allow for greater transaction concurrency than the protected ready modes. The protected ready modes, on the other hand, create less overhead than the shared ready modes and reduce the chances for deadlocking.

In exclusive ready mode, as in the protected ready modes, CA IDMS provides protection at the area level. However, exclusive ready mode prohibits other transactions from retrieving data from the area.

Ready Modes and Later Modifications

The ready clause only affects areas accessed by statements compiled at the time the CREATE ACCESS MODULE statement is issued. If new areas are added at a later time because the access module is altered or because dynamic SQL accesses additional areas at runtime, those areas are accessed using a ready mode determined by the above rules as if no READY option had been specified (unless the READY option is repeated on the ALTER ACCESS MODULE statement).
Example

Creating an Access Module

The following CREATE ACCESS MODULE statement creates an access module from seven RCMs. The schema name EMP_TEST is replaced with EMP_PROD when it qualifies a table or view name, and unqualified tables and views are assumed to be in the EMP_PROD schema.

By default, CA IDMS performs the following tasks:

- Checks the definition timestamps for all tables in the access module before executing the first statement and automatically re-creates the access module if any timestamps do not match
- Places a shared update lock on each area in the access module at the time of the first request for data in the area

The following example shows creating an access module.

```
create access module hrprod.empam001
   from emp_dict.empdsp01,
     emp_dict.empdsp02,
     emp_dict.empdsp03,
     emp_dict.empadd01,
     emp_dict.empupd01,
     emp_dict.empupd02,
     emp_dict.empdel01
   map emp_test to emp_prod,
     null to emp_prod;
```

More Information

- For more information about ACCESS modules, see ALTER ACCESS MODULE and DROP ACCESS MODULE (see page 418) or see the CA IDMS Database Administration Guide.
- For more information about schema-name mappings, see Identifying Entities in Schemas.
- For more information about specifying isolation level, see SET TRANSACTION or see the CA IDMS SQL Programming Guide.
- For more information about ready modes, see the CA IDMS Database Administration Guide.
CREATE CALC

The CREATE CALC data description statement defines a CALC key on a base table. The CALC key definition is stored in the dictionary. It is also a CA IDMS extension of the SQL standard. You can define only one CALC key on any given table.

Authorization

To issue a CREATE CALC statement, you must own or have the ALTER privilege on the table on which the CALC key is being defined.

Syntax

```
CREATE CALC key
  ON schema-name. table-identifier
  ( column-name )
```

Parameters

UNIQUE

Specifies the CALC key value in any given row of the table on which the CALC key is being defined must be different from the CALC key value in any other row of the table. A table with a unique CALC key cannot contain duplicate CALC key values.

**table-identifier**

Specifies the table on which the CALC key is being defined. Table-identifier must identify a base table defined in the dictionary. The named table cannot:

- Contain any data
- Have a clustered index defined on it
- Be the referencing table in a clustered referential constraint

**schema-name**

Identifies the schema associated with the named table.

If you do not specify schema-name, it defaults to:

- The current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically
- The schema associated with the access module used at runtime, if the statement is embedded in an application program
(column-name)

Specifies one or more columns that make up the CALC key. Column-name must identify a column in the table on which the CALC key is being created and must be unique within the list of column names.

You can include from 1 through 32 columns in a CALC key.

Usage

SYSTEM tables

You cannot define a CALC key on a table in the SYSTEM schema.

Example

Defining a Unique CALC Key

The following CREATE CALC statement defines a unique CALC key on the COVERAGE table. The CALC key consists of two columns: PLAN_CODE and EMP_ID.

create unique calc key
  on coverage
    (plan_code, emp_id);

Note: For more information about dropping CALC key definitions, see DROP CALC.

CREATE CONSTRAINT

The CREATE CONSTRAINT data description statement defines a referential constraint in the dictionary. A referential constraint establishes a relationship between two tables.

Using the CREATE CONSTRAINT statement, you can also specify how the constraint is implemented physically. It is also a CA IDMS extension of the SQL Standard.

Authorization

To issue a CREATE CONSTRAINT statement, you must:

- Either hold the ALTER privilege on or own the referencing table in the constraint being defined
- Hold the REFERENCES privilege on the referenced table in the constraint being defined
Syntax

CREATE CONSTRAINT constraint-name

    schema-name. referencing-table ( foreign-key-column )

    REFERENCES schema-name. referenced-table

    ( referenced-column )

    LINKED linked-constraint-options

    UNLINKED

Expansion of linked-constraint-options

    CLUSTERED

    INDEX

    COMPRRESSED

    UNCOMPRESSED

    index-block-specification

    ORDER BY ( sort-column ASC DESC)

    UNIQUE

Expansion of index-block-specification

    INDEX BLOCK contains key-count keys

    DISPLACEMENT IS page-count PAGES

Parameters

constraint-name

Specifies the name of the referential constraint being created. Constraint-name
must be a 1- through 18-character name that follows the conventions for SQL
identifiers.

Constraint-name must be unique for the schema of the referencing table.

referencing-table

Specifies the referencing table in the constraint. Referencing-table must identify a
base table defined in the dictionary.

If you specify CLUSTERED in the CREATE CONSTRAINT statement, referencing-table:

■ Cannot have a CALC key or clustered index defined on it
■ Cannot be the referencing table in another clustered constraint

If you specify LINKED in the CREATE CONSTRAINT statement, referencing-table:

■ Must be empty
■ Must not be the same table as referenced-table
**CREATE CONSTRAINT**

- **schema-name**
  - Identifies the schema associated with the referencing table.
  - If you do not specify `schema-name`, it defaults to:
    - The current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically
    - The schema associated with the access module used at runtime, if the statement is embedded in an application program

- **(foreign-key-column)**
  - Specifies one or more columns that make up the foreign key in the referencing table. `Foreign-key-column` must identify a column in the referencing table and must be unique within the list of column names.
  - If you specify UNLINKED in a `CREATE CONSTRAINT` statement (or accept UNLINKED as the default), the foreign key must be a CALC key or an index key, as defined by a `CREATE CALC` or `CREATE INDEX` statement.
  - You can include from 1 through 32 columns in a foreign key.

- **REFERENCES referenced-table**
  - Specifies the referenced table in the constraint. `Referenced-table` must identify a base table defined in the dictionary.
  - If you specify LINKED in a `CREATE CONSTRAINT` statement, `referenced-table`:
    - Must be empty
    - Must not be the same table as `referencing-table`

- **schema-name**
  - Identifies the schema associated with the referenced table.
  - If you do not specify `schema-name`, it defaults to:
    - The current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically
    - The schema associated with the access module used at runtime, if the statement is embedded in an application program
(referenced-column)

Specifies one or more non-null columns that make up a unique key in the referenced table, as defined by a CREATE CALC or CREATE INDEX statement.

Referenced-column must identify a column in the referenced table and must be unique within the list of column names. The columns must be named in the CREATE CONSTRAINT statement in the same order in which they are named in the CREATE CALC or CREATE INDEX statement that defines the unique key.

You must specify the same number of referenced columns as the number of columns included in the foreign key of the referencing table. The corresponding referenced and foreign-key columns must have the same data type, length, precision, and scale.

LINKED

Directs CA IDMS to maintain a physical linkage between the rows in the referenced and referencing tables.

linked-constraint-options

Specifies additional characteristics of a linked constraint. Expanded syntax for linked-constraint-options is shown immediately following the CREATE CONSTRAINT syntax.

UNLINKED

Directs CA IDMS not to physically link the referenced and referencing tables.

If you specify UNLINKED, the referencing table must have a CALC key or index defined on the foreign key and the order of columns of the CALC or index key must match the order of columns of the foreign key. The index or CALC key on the foreign key does not have to be unique.

A constraint in which a single table is the referencing table and the referenced table must be unlinked.

UNLINKED is the default when you specify neither LINKED nor UNLINKED.

Note: If you are using an index, it can contain additional columns that are not part of the foreign key. The foreign key columns must precede any additional columns in the index key.

CLUSTERED

Specifies each row of the referencing table is to be stored close to other rows of the referencing table that have the same non-null foreign-key value.
CREATE CONSTRAINT

Parameters for Expansion of linked-constraint-options

INDEX

Directs CA IDMS to create an index between the referenced and referencing tables.

COMPRESSED

Directs CA IDMS to maintain index entries in a compressed form in the database.

UNCOMPRESSED

Directs CA IDMS to maintain index entries in an uncompressed form in the database.

UNCOMPRESSED is the default when you specify neither COMPRESSED nor UNCOMPRESSED.

index-block-specification

Establishes characteristics of the index created between the referenced and referencing tables.

Syntax for index-block-specification follows the syntax for linked-constraint-options.

ORDER BY (sort-column)

Specifies one or more columns that make up a sort key for a linked constraint. CA IDMS uses the sort key to determine the order in which the rows of the referencing table are to be linked within the referential constraint. Rows are linked in ascending or descending order, first by the first column specified, then by the second column specified within the ordering established by the first column, then by the third column specified, and so on.

Sort-column must identify a column in the referencing table and must be unique within the list of column names.

If you specify the UNIQUE option of the ORDER BY parameter, each column included in the sort key must be defined as NOT NULL.

You can specify a maximum of 32 sort columns.

ASC

Indicates that values in the named column are to be ordered in ascending sequence. ASC is the default when you specify neither ASC nor DESC.

DESC

Indicates that values in the named column are to be ordered in descending sequence.
UNIQUE

Specifies the sort-key value in any given row of the referencing table must be different from the sort-key value in any other row of the table that has the same non-null foreign-key value. A table with a unique sort key cannot contain duplicate rows.

Parameters for Expansion of index-block-specification

key-count KEYS

Establishes the maximum number of entries in each internal index record (SR8 system record).

Key-count must be an unsigned integer in the range 3 through 8130.

If you do not specify KEYS, key-count defaults to 10.

page-count PAGES

Indicates how far away from the referenced row the bottom-level index records are to be stored.

Page-count must be an unsigned integer in the range 0 through 32767.

If index-block-specification is omitted, the value of page-count is 0.

If the value of page-count is 0, the bottom-level internal index records cannot be displaced from the referenced row with which they are associated.

Usage

System-owned Tables

You cannot define a referential constraint where the referencing table or the referenced table is in the SYSTEM schema.

Specifying a Linked Constraint

A linked constraint (as opposed to an unlinked constraint) is used by the optimizer in determining the most efficient access for an SQL DML statement. It does not affect either the syntax or the semantics of the statement.

Dropping Tables

When you define a referential constraint, you restrict the conditions under which tables can be dropped.

Mixed Page Group

A constraint defined as linked clustered can not span page groups. The referencing and referenced tables of a constraint defined as linked clustered must be in the same page group.
Examples

Defining a self-referencing Constraint

The following CREATE CONSTRAINT statement defines a referential constraint in which the EMPLOYEE table is the referencing and the referenced table. This constraint directs CA IDMS to ensure that the value in the MANAGER_ID column in each row of the EMPLOYEE table matches the value in the EMP_ID column in another row of the table. By default, the constraint is unlinked. (Self-referencing constraints must be unlinked.)

```
create constraint manager_emp
  employee
  (manager_id)
  references employee
  (emp_id);
```

Defining a Linked Constraint

The following CREATE CONSTRAINT statement defines a referential constraint between the BENEFITS table and the EMPLOYEE table. This constraint directs CA IDMS to ensure that the value in the EMP_ID column in each row of the BENEFITS table matches the value in the EMP_ID column in a row of the EMPLOYEE table. The referential constraint is implemented with a linked index, with the index entries sorted in descending order by the value in the FISCAL_YEAR column.

```
create constraint emp_benefits
  benefits
  (emp_id)
  references employee
  (emp_id)
  linked index
  order by (fiscal_year desc);
```

More Information

- For more information about dropping referential constraints, see DROP CONSTRAINT.
- For more information about defining CALC keys, see CREATE CALC.
- For more information about defining indexes, see CREATE INDEX.
- For more information about implementing referential constraints, see the CA IDMS Database Design Guide.
- For more information about dropping tables, see DROP TABLE.
CREATE FUNCTION

The CREATE FUNCTION data description statement stores the definition of a function in the SQL catalog. You can then invoke the function in any value-expression of an SQL statement except in the search condition of a table's check constraint. The function invocation results in CA IDMS calling the corresponding routine. Such routines can perform any action and return a single scalar value. You use the formal parameters of a function definition to specify the data type and format of the data to be passed to the function. Similarly, the data type of the return value is specified in the function definition.

Functions can be defined with a language of SQL, in which case, the routine actions written as SQL statements are specified and stored together with the function definition in the SQL catalog.

Authorization

To issue a CREATE FUNCTION statement, you must own the schema in which the function is being defined or hold the CREATE privilege on the named function.

Syntax

```sql
CREATE FUNCTION function-identifier
    (parameter-definition) RETURNS data-type
    EXTERNAL NAME external-routine-name
    language-clause
    PROTOCOL IDMS ADS
    ESTIMATED ROWS row-count
    ESTIMATED IOS io-count
    USER MODE
    SYSTEM MODE
    LOCAL WORK AREA local-stge-size
    GLOBAL WORK AREA global-stge-size
    KEY key-id
```
CREATE FUNCTION

TRANSACTION SHARING ON OFF DEFAULT

DEFAULT DATABASE NULL CURRENT

TIMESTAMP timestamp-value

procedure-statement

ADS COMPILE OPTION → compile-option ↓

Establishment of parameter-definition

parameter-name → data-type ↓ WITH DEFAULT ↓

Establishment of language-clause

LANGUAGE

ADS ASSEMBLER COBOL PLI SQL

Establishment of procedure-statement

SQL-AM-mgmt-stmt
SQL-authorization-stmt
SQL-Control-stmt
SQL-Diagnostics-stmt
SQL-DML-stmt
SQL-DDL-stmt
SQL-transaction-stmt

Parameters

function-identifier

Specifies the 1- to 18-character name of the function you are creating.

Function-identifier must:

■ Be unique among the function, table, table procedure, procedure and view identifiers within the schema associated with the function

■ Follow conventions for SQL identifiers

schema-name

Specifies the schema name qualifier to be associated with the function.

Schema-name must identify a schema defined in the dictionary. If you do not specify a schema-name, it defaults to:

■ The current schema associated with your SQL session, if the statement is specified through the Command Facility or executed dynamically

■ The schema associated with the access module used at runtime, if the statement is embedded in an application program
**CREATE FUNCTION**

Chapter 8: Statements

**parameter-definition**

Defines a parameter to be associated with the function. Parameters pass to the function in the order you specify them. You must enclose the list of parameters in parentheses. You must separate multiple parameter definitions by commas.

Expanded syntax for **parameter-definition** is shown above immediately following the CREATE FUNCTION syntax. Descriptions for these parameters are located at the end of this section.

**RETURNS data-type**

Specifies the data type of the returned value. The returned value is implicitly nullable and can be set to NULL in the external routine. The returned value is accessible to the external routine as an extra parameter with the implicit name USER_FUNC, which comes immediately after the function parameters.

**external-routine-name**

Specifies the one- to eight-character name of the program or mapless dialog that CA IDMS calls to process function invocation.

For functions written in SQL, the external-routine-name should specify a name that is unique within the dictionary that holds the function definition. In other words, the name should be different from any other external name of any SQL-invoked routine and from any CA ADS dialog, RCM, or AM name.

**row-count**

Specifies an integer value, in the range 0 through 2,147,483,647, representing the average number of rows the CA IDMS optimizer uses for cost calculation of the function invocation.

**io-count**

Specifies an integer value, in the range 0 through 2,147,483,647, representing the average number of disk accesses generated by the function for a given set of input parameters.

**language-clause**

Specifies the programming language of the function. This clause is required for functions written in SQL. For others, it is documentational only. If the language is not specified, it is treated as null. Expanded syntax for **language-clause** is shown above immediately following the CREATE FUNCTION syntax. Descriptions for these parameters are located at the end of this section.
PROTOCOL

Specifies the protocol with which the function is invoked. This specification is required except with language SQL. If LANGUAGE SQL is specified, PROTOCOL must be ADS or the clause must not be specified.

IDMS

Use IDMS for functions that are written in COBOL, PL/I, or Assembler.

ADS

Use ADS for functions that are written in CA ADS. The name of the dialog that is loaded and executed when the function is invoked is specified by the external-routine-name in the EXTERNAL NAME clause. ADS is the default if LANGUAGE SQL is specified.

USER MODE

Specifies that the function should execute as a user-mode application program within CA IDMS. This cannot be specified with language SQL or protocol ADS. For other languages and protocols, it is the default.

SYSTEM MODE

Specifies that the function should execute as a system-mode application program. SYSTEM MODE is the default if language is SQL.

To execute as a system mode application, the program must be fully reentrant and be written in either:

- ADS as a mapless dialog
- Assembler using DC calling conventions
- COBOL or PL/I and compiled with an LE-compliant compiler.

Note: If protocol is set to ADS, you must specify MODE SYSTEM.
**local-stge-size**

Specifies an integer value, in the range 0 through 32767, representing the size, in bytes, of a local storage area that CA IDMS allocates at runtime and passes to the function on each invocation.

**global-stge-size**

Specifies an integer value, in the range 0 through 32767, representing the size, in bytes, of the global storage area that CA IDMS allocates at runtime and passes to the function on each invocation.

CA IDMS allocates a global storage area once within a transaction and retains it until the transaction terminates.

**key-id**

Specifies the one- to four-character identifier for the global storage area. CA IDMS passes the same piece of global storage within a transaction to all routines that have the same global storage key.

If you do not specify a storage key, CA IDMS allocates each function its own global storage area, which will not be used for any other routine within the transaction.

**TRANSACTION SHARING**

Specifies whether transaction sharing should be enabled for database sessions started by the function. If transaction sharing is enabled for a function's database session, it shares the current transaction of the SQL session. If language SQL is specified, TRANSACTION SHARING must be ON or the clause must not be specified.

**ON**

Specifies that transaction sharing should be enabled. ON is the default if language is SQL.

**OFF**

Specifies that transaction sharing should be disabled.

**DEFAULT**

Specifies that the transaction sharing setting in effect when the function is invoked should be retained. DEFAULT is the default for languages other than SQL.
CREATE FUNCTION

compile-option

Specifies a CA ADS option to be used when compiling the dialog associated with an SQL function. The options that can be specified and the syntax to use are given in the CA ADS Reference Guide, Appendix D.2.6 Dialog-expression. Compile-option can be specified only if language is SQL.

Note: The ability to specify the ADS COMPILE OPTION clause is a CA IDMS extension.

procedure-statement

Specifies the actions taken in the function. Procedure-statement is required if language is SQL. It cannot be specified otherwise. Expanded syntax for procedure-statement is shown above immediately following the CREATE FUNCTION syntax. Descriptions for these parameters are located at the end of this section.

DEFAULT DATABASE

Specifies whether a default database should be established for database sessions started by the function.

NULL

Specifies that no default database should be established.

CURRENT

Specifies that the database to which the SQL session is connected should become the default for any database session started by the function.

timestamp-value

Specifies the value of the synchronization stamp to be assigned to the function. Timestamp-value must be a valid external representation of a timestamp.

Parameters for Expansion of parameter-definition

parameter-name

Specifies a 1- to 32-character name of a parameter that passes to the function. Parameter-name must:

■ Be unique within the function that you are defining
■ Follow the conventions for SQL identifiers

All parameters are implicitly nullable and thus can be assigned NULL as a parameter value.
**data-type**

Defines the data type for the named parameter. For expanded data-type syntax, see Expansion of Data-type.

**WITH DEFAULT**

Directs CA IDMS to pass a default value for the named parameter if you do not specify a value for the function invocation.

The default value for a parameter is based on its data type:

<table>
<thead>
<tr>
<th>Column data type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>Blanks</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>A character string literal with a length of zero (that is, '')</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>Double-byte blanks</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>A double-byte character string literal with a length of zero</td>
</tr>
<tr>
<td>DATE</td>
<td>The value in the CURRENT DATE special register</td>
</tr>
<tr>
<td>TIME</td>
<td>The value in the CURRENT TIME special register</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>The value in the CURRENT TIMESTAMP special register</td>
</tr>
<tr>
<td>All numeric data types</td>
<td>0 (zero)</td>
</tr>
</tbody>
</table>

**Parameters for Expansion of language-clause**

**ADS**

Specifies that the SQL routine is written in the CA ADS language.

**ASSEMBLER**

Specifies that the SQL routine is written in the assembler language.

**COBOL**

Specifies that the SQL routine is written in the COBOL language.

**PLI**

Specifies that the SQL routine is written in the PL/I language.

**SQL**

Specifies that the SQL routine is written in the SQL language.

**Note:** The ability to specify ADS or ASSEMBLER as a language is a CA IDMS extension.
Parameters for Expansion of procedure-statement

**SQL-AM-mgmt-stmt**
Specifies a statement from the Access Module Management Statements category.

**SQL-authorization-stmt**
Specifies a statement from the Authorization Statements category.

**SQL-Control-stmt**
Specifies a statement from the Control Statements category.

**SQL-Diagnostics-stmt**
Specifies a statement from the Diagnostics Statements category.

**SQL-DDL-stmt**
Specifies a statement from the Data Description Statements category.

**SQL-DML-stmt**
Specifies a statement from the Data Manipulation Statements category.

**SQL-session-mgmt-stmt**
Specifies a statement from the Session Management Statements category.

**Note:** The ability to include a RELEASE, SUSPEND, or RESUME statement in an SQL routine is a CA IDMS extension.

**SQL-transaction-mgmt-stmt**
Specifies a statement from the Transaction Management Statements category.

**Note:** The ability to include a COMMIT or ROLLBACK statement in an SQL routine is a CA IDMS extension.
Usage

Coding functions with language SQL

The rules for coding the procedure body of an SQL function are given by

**procedure-statement**. A procedure body typically contains multiple SQL statements and
according to the SQL grammar, SQL statements are terminated by the semi-colon.

However, to define SQL routines, the Command Facility (OCF, IDMSBCF, or Visual DBA
OCF console) needs to be used. It also has the semi-colon as the default command
terminator. Before a new command can be specified, the CREATE FUNCTION needs to
be terminated by a semi-colon. Clearly, the semi-colon cannot concurrently be used as a
terminator by both the SQL procedure language and the Command Facility. Therefore,
when **procedure-statement** contains multiple SQL statements or when the ADS
COMPILE OPTION is specified, the Command Facility needs to use a terminator different
from the semi-colon. To accomplish this, a SET OPTIONS COMMAND DELIMITER
'delimiter-string' must be executed. Changing the terminator of the Command Facility
remains in effect until the end of the session or until a new SET OPTIONS COMMAND
DELIMITER is encountered. For more information about SET OPTIONS, see the
Command Facility chapter in the *CA IDMS Common Facilities Guide*.

Language SQL

If LANGUAGE SQL is specified, the following attribute settings are established by default
and must not be overridden to a different value:

- Protocol is ADS
- Mode is SYSTEM
- Transaction sharing is ON

Functions whose language is SQL are implemented through an automatically generated
CA ADS dialog whose name is *external-routine-name*.

An error while parsing **procedure-statement** or an error while compiling the associated
CA ADS dialog causes termination of the CREATE FUNCTION statement with a warning
instead of a statement error. This allows the erroneous **procedure-statement** syntax to
be saved in the catalog for later correction using the DISPLAY FUNCTION command. The
CA ADS dialog and associated access module are not created.

Specifying CA ADS Compile Options

If LANGUAGE SQL is specified, you can specify one or more compile options to be used
when the associated dialog is compiled. Specifying compile options can be useful for
debugging purposes to enable tracing and the use of online debugging facilities. Compile
options can also be used to include additional work records and SQL tables which can be
referenced in native CA ADS code included in the routine body.
Some useful compile options include:

- SYMBOL TABLE IS YES - to allow the use of symbols by the TRACE command and the online debug facilities
- ADD RECORD record-name - to enable manipulation of elements from the specified record
- ADD SQL TABLE table-name - to enable manipulation of columns or parameters of the specified SQL table-like object.

**Specifying a Synchronization Stamp**

When defining or altering a function, you can specify a value for its synchronization stamp. You should use care when doing so because the purpose of the stamp is to enable the detection of discrepancies between an entity and its definition. If explicitly specified, you must set the synchronization stamp to a new value following a change so that the change is detectable by the runtime system.

If not specified, the synchronization stamp is automatically set to the current date and time.

**Grouping Procedure Statements into a Single Statement**

Multiple procedure statements can be grouped together as a compound statement. A compound statement is a control statement and therefore it is also a procedure statement.
Examples

Example for CREATE FUNCTION

```sql
CREATE FUNCTION FIN.UDF_FUNBONUS
  ( F_EMP_ID        DECIMAL(4)     )
  RETURNS DECIMAL(10)
  EXTERNAL NAME FUNBONUS PROTOCOL IDMS
  DEFAULT DATABASE CURRENT
  USER MODE
  LOCAL WORK AREA 0
 ;
```

Example for Language SQL

```sql
set options command delimiter '++';
drop function USER01.TCNTEQNAME++
commit++
create function USER01.TCNTEQNAME
  ( TITLE     varchar(40) with default
    , P_FNAME   char(20)
    , P_COUNT     integer
    , RESULT    varchar(10)
  ) RETURNS   varchar(20)
  EXTERNAL NAME TCNTEQN LANGUAGE SQL

Label_700:
begin not atomic
  /*
   ** Count number of employees with equal Firstname
   */
  declare FNAME       char(20);
  declare LNAME       varchar(20);
  declare P_COUNT_SAV integer default 0;

  declare EMP1 CURSOR FOR
    Select EMP_FNAME, EMP_LNAME
    From DEMOEMPL.EMPLOYEE
    where EMP_FNAME = P_FNAME;
  open EMP1;
  fetch EMP1 into FNAME, LNAME;
```
The CREATE INDEX data description statement defines an index on a base table. The index definition is stored in the dictionary. It is also a CA IDMS extension of the SQL standard.

**Authorization**

To issue a CREATE INDEX statement, you must:
- Hold the ALTER privilege on or own the table on which the index is being defined
- Hold the USE privilege on the area where the named index is stored
Syntax

```
CREATE UNIQUE INDEX index-name
  ON schema-name. table-identifier
  (column-name ASC | DESC)
  COMPRESSED | UNCOMPRESSED
  IN segment-name.area-name index-block-specification
  CLUSTERED
  INDEX ID index-id-number
```

Expansion of index-block-specification

```
INDEX BLOCK CONTAINS key-count KEYS
  DISPLACEMENT IS page-count PAGES
```

Parameters

**UNIQUE**

Specifies that the index-key value in any given row of the table on which the index is being defined must be different from the index-key value in any other row of the table. A table with a unique index cannot contain duplicate index key values.

If you specify UNIQUE, and the table on which the index is being defined contains duplicate rows, CA IDMS returns an error.

**index-name**

Specifies the name of the index being created. *Index-name* must be a 1- through 18-character name that follows the conventions for SQL identifiers.

*Index-name* must be unique for the table on which the index is defined.

**ON table-identifier**

Specifies the table on which the index is being defined. *Table-identifier* must identify a base table defined in the dictionary.

If you specify CLUSTERED in a CREATE INDEX statement, the named table:

- Cannot have a CALC key defined on it
- Cannot have another clustered index defined on it
- Cannot be the referencing table in a clustered referential constraint
**schema-name**

Identifies the schema associated with the named table.

If you do not specify *schema-name*, it defaults to:

- The current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically
- The schema associated with the access module used at runtime, if the statement is embedded in an application program

**(column-name)**

Specifies one or more columns that make up the index key. CA IDMS maintains index entries in ascending or descending order according to the values in the specified columns. Entries are ordered first by the first column specified, then by the second column specified within the ordering established by the first column, then by the third column specified, and so on.

*Column-name* must identify a column in the table on which the index is being created and must be unique within the list of column names.

You can include from 1 through 32 columns in an index key.

If no column name is specified, CA IDMS creates an index on the db-key sorted in ascending order.

**ASC**

Indicates that values in the named column are to be sorted in ascending order. ASC is the default when you specify neither ASC nor DESC.

**DESC**

Indicates that values in the named column are to be sorted in descending order.

**COMPRESSED**

Directs CA IDMS to maintain index entries in a compressed form in the database.

**UNCOMPRESSED**

Directs CA IDMS to maintain index entries in an uncompressed form in the database.

UNCOMPRESSED is the default when you specify neither COMPRESSED nor UNCOMPRESSED.
IN

Specifies the area to be used to store entries in the index.

If you do not associate an area with an index, CA IDMS uses the area associated with the table on which the index is being defined.

segment-name

Identifies the segment associated with the area.

area-name

Identifies the area to be associated with the index. Area-name must identify an area defined in the dictionary.

index-block-specification

Establishes characteristics of the index.

Syntax for index-block-specification immediately follows the syntax for CREATE INDEX.

If index-block-specification is omitted, values for key-count and page-count are calculated by CA IDMS using available information about actual or estimated row count for the table on which the index is being defined.

CLUSTERED

Specifies that each row of the table on which the index is being defined is to be stored as close as possible to the table row with the immediately preceding index-key value.

INDEX ID index-id-number

Assigns an index ID value for the index being created. The index-id-number must be in the range of 1 through 32767.

Parameters for Expansion of index-block-specification

key-count KEYS

Establishes the maximum number of entries in each internal index record (SR8 system record).

Key-count must be an unsigned integer in the range 3 through 8180.

page-count PAGES

Indicates how far away from the top of the index (the SR7 system record) the bottom-level index records are to be stored.

Page-count must be an unsigned integer in the range 0 through 32767.

If the value of key-count is 0, the bottom-level internal index records are not displaced from the SR7 record.
Usage

Specifying an Index ID

When defining an index you can specify a value for its numeric index identifier. If explicitly specified, it must be unique across all other indexes residing in the same database area. If not specified, the index's numeric identifier is automatically set to the next available number in the range 1 through 32,767.

SYSTEM Tables

You cannot define an index on a table in the SYSTEM schema.

SYSTEM Areas

You cannot associate an index with a system area supplied with CA IDMS.

Order of Null Values

If the value of an index key column is null, it is treated as higher than all non-null values.

Null Values in Unique Indexes

Nullable columns are allowed to be used in a UNIQUE index. Null values are treated like any other value when the uniqueness of an index is evaluated. For example, a single column index can only contain one null value.

Mixed Page Group

An index must reside in the same page group as the table on which the index is created.
Example

Defining a Unique Index

The following CREATE INDEX statement defines a unique index on the JOB table. The index key consists of two columns: JOB_ID and JOB_TITLE. The index entries are stored in compressed form in the same area as the JOB table.

```
cREATE UNIQUE INDEX job_title_index
    ON job
    (job_id, job_title)
    COMPRESSED;
```

Defining a Clustered Index

The following CREATE INDEX statement defines an index on the MONTHLY_BUDGET table. The index key consists of two columns: FISCAL_YEAR and MONTH. The index entries are stored in compressed form in the SALESSEG.SALES_X_AREA area. Rows of the MONTHLY_BUDGET table that have consecutive index-key values are stored close to each other.

```
CREATE INDEX budget_date_index
    ON sales_sch.monthly_budget
    (fiscal_year desc, month)
    COMPRESSED
    IN salesseg.sales_x_area
    CLUSTERED;
```

More Information

- For more information about dropping indexes, see DROP INDEX.
- For more information about implementing indexes, see the CA IDMS Database Design Guide.
- For more information about calculations, see Error! Reference source not found..

More information:

DROP INDEX (see page 424)

CREATE KEY

The CREATE KEY statement defines a key on a procedure or table procedure. The key definition is stored in the dictionary. It is also a CA IDMS extension of the SQL standard.
Authorization

To issue a CREATE KEY statement, you must own or hold the ALTER privilege on the procedure or table procedure on which the key is being defined.

Syntax

```sql
CREATE KEY key-name
UNIQUE | PRIMARY
ON schema-name.
procedure-identifier
| table-procedure-identifier
(
| parameter-name.
) 
ESTIMATED ROWS row-count
ESTIMATED IOS io-count
```

Parameters

**UNIQUE**

Specifies the key value is unique to a row that the procedure or table procedure returns. CA IDMS does not enforce this restriction. The procedure or table procedure itself must enforce uniqueness.

**PRIMARY**

Specifies the key is unique and that it is the most commonly-used key for identifying specific rows returned by the procedure. While you can define several unique keys for a procedure or table procedure, you can specify only one primary key.

**key-name**

Specifies the name of the key. The `key-name` must be:

- A 1- to 18-character name that follows the conventions for SQL identifiers
- Unique for the procedure or table procedure on which the key is defined
ON table-procedure-identifier

Specifies the table procedure for which you are defining the key. The table-procedure-identifier must identify a table procedure defined in the dictionary.

procedure-identifier

Specifies the procedure for which you define the key. The procedure-identifier must identify a procedure defined in the dictionary.

schema-name

Identifies the schema associated with the procedure or table procedure.

If you do not specify a schema-name it defaults to:

- The current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically
- The schema associated with the access module used at runtime, if the statement is embedded in an application program

(parameter-name)

Specifies one or more procedure or table procedure parameters that form the key. The parameter-name must:

- Identify a parameter of the procedure or table procedure on which the key is defined
- Be unique within the list of parameter names

You can include as many as 32 parameters in a key.

row-count

Specifies an integer value, in the range of 0 through 2,147,483,647, which represents the number of rows that the procedure or table procedure returns when input values are provided for all the parameters in the key.

io-count

Specifies an integer value, in the range of 0 through 2,147,483,647, which represents the number of disk accesses that the procedure or table procedure generates while returning row-count rows when input values are provided for all the parameters in the key.
Usage

Enforcing Uniqueness

It is the responsibility of the procedure or table procedure to enforce the uniqueness of the procedure or table procedure keys; for example, on an INSERT into a table procedure, CA IDMS makes no attempt to determine whether a duplicate row, with respect to a unique table procedure key, exists. The table procedure, in conjunction with database services it invokes, is responsible for ensuring uniqueness.

Influencing Join Strategies

CA IDMS uses procedure or table procedure key information when determining the best approach to satisfy queries that join procedure or table procedures with other tables, views, procedures or table procedures. Specifically, if the set of column values provided on a particular call to the table procedure matches the columns defined in the table procedure’s KEY, the ESTIMATED ROWS and ESTIMATED I/Os for that KEY are used during optimization. If these statistics are provided, and data is passed to the table procedure’s key by the WHERE clause during execution, the optimizer uses the statistical information when the table procedure is joined with other tables or views. Providing estimated-row and I/O counts, for the procedure or table procedure and for each access key that the procedure uses, allows CA IDMS to select the optimal access strategy.

Unique Keys for CA IDMS Server

If you define procedure or table procedure keys, CA IDMS Server reports this information when processing an ODBC request to return key information for a procedure. The ability to return key information is particularly important for certain ODBC-based products which require a unique key to update and delete data.

Example

The following CREATE KEY statements define three keys on the EMP.ORG table procedure. The first two keys are simple access keys; the third defines a primary key for CA IDMS Server to use.

(1) CREATE KEY ORG1 ON EMP.ORG (EMP_ID)
    ESTIMATED ROWS 3
    ESTIMATED I/Os 3;

(2) CREATE KEY ORG2 ON EMP.ORG (MGR_ID)
    ESTIMATED ROWS 5
    ESTIMATED I/Os 5;

(3) CREATE PRIMARY KEY ORG3 ON EMP.ORG
    (MGR_ID, EMP_ID, START_DATE)
    ESTIMATED ROWS 1
    ESTIMATED I/Os 3;
More Information

- For more information about influencing join strategies, see Defining and Using Table Procedures.
- For more information about defining table procedures, see CREATE TABLE PROCEDURE.
- For more information about dropping keys, see DROP KEY.
- For more information about defining procedures, see CREATE PROCEDURE.
- For more information about the WHERE clause, see WHERE Clause References.

CREATE PROCEDURE

The CREATE PROCEDURE data description statement stores the definition of a procedure in the SQL catalog. You can refer to the procedure in an SQL CALL statement or in an SQL SELECT statement just as you would a table procedure. These references result in CA IDMS calls to the corresponding routine. Such routines can perform any action, such as manipulating data stored in some other organization (for example, in a non SQL-defined database or in a set of VSAM files). You can also use them to implement business logic.

Procedures can be defined with a language of SQL. The routine actions, written as SQL statements, are specified and stored together with the procedure definition in the SQL catalog.

The formal parameters of a procedure definition can be used like columns of a table during a procedure invocation to pass values to and from the procedure.

Authorization

To issue a CREATE PROCEDURE statement, you must own the schema in which the procedure is being defined or hold the CREATE privilege on the named procedure.

Syntax

```
CREATE PROCEDURE procedure-identifier

(parameter-definition) EXTERNAL NAME external-routine-name

language-clause

PROTOCOL IDMS

ESTIMATED ROWS row-count

ESTIMATED IOS io-count
```
CREATE PROCEDURE

- USER MODE
- SYSTEM MODE

- LOCAL WORK AREA  local-stge-size
- GLOBAL WORK AREA  global-stge-size
  - KEY key-id

- TRANSACTION SHARING
  - ON
  - OFF
  - DEFAULT

- DEFAULT DATABASE
  - NULL
  - CURRENT

- TIMESTAMP  timestamp-value

- DYNAMIC RESULT SETS maximum-dynamic-result-sets

- ADS COMPILe OPTION  compile-option :

**Expansion of parameter-definition**

- parameter-name  data-type  WITH DEFAULT

**Expansion of language-clause**

- LANGUAGE
  - ADS
  - ASSEMBLER
  - COBOL
  - PLI
  - SQL

**Expansion of procedure-statement**

- SQL-AM-mgmt-stmt
- SQL-authorization-stmt
- SQL-Control-stmt
- SQL-Diagnostics-stmt
- SQL-DML-stmt
- SQL-transaction-mgmt-stmt
Parameters

**procedure-identifier**

Specifies the 1- to 18-character name of the procedure you are creating. *Procedure-identifier* must:

- Be unique among the function, procedure, table, table procedure and view identifiers within the schema associated with the procedure
- Follow conventions for SQL identifiers

**schema-name**

Specifies the schema name qualifier to be associated with the procedure. *Schema-name* must identify a schema defined in the dictionary. If you do not specify a *schema-name*, it defaults to:

- The current schema associated with your SQL session, if the statement is specified through the Command Facility or executed dynamically
- The schema associated with the access module used at runtime, if the statement is embedded in an application program

**parameter-definition**

Defines a parameter to be associated with the procedure. Parameters pass to the procedure in the order you specify them. You must enclose the list of parameters in parentheses. You must separate multiple parameter definitions by commas.

Expanded syntax for **parameter-definition** is shown above immediately following the CREATE PROCEDURE syntax. Descriptions for these parameters are located at the end of this section.

**external-routine-name**

Specifies the one- to eight-character name of the program which is called to process references to the procedure.

For procedures written in SQL, the external-routine-name should specify a name that is unique within the dictionary that holds the procedure definition. In other words, the name should be different from any other external name of any SQL-invoked routine and from any &USIDCADS. dialog, RCM, or AM name.

**language-clause**

Specifies the programming language of the procedure. This clause is required for procedures written in SQL. For others, it is documentational only. If the language is not specified, it is treated as null.

**PROTOCOL**

Specifies the PROTOCOL with which the procedure is invoked. This specification is required except with language SQL. If LANGUAGE SQL is specified, PROTOCOL must be ADS or the clause must not be specified.
IDMS

Use IDMS for procedures that are written in COBOL, PL/I, or Assembler.

ADS

Use ADS for procedures that are written in CA ADS. The name of the dialog that is loaded and executed when the procedure is invoked is specified by the external-routine-name in the EXTERNAL NAME clause. ADS is the default if LANGUAGE SQL is specified.

row-count

Specifies an integer value, in the range 0 through 2,147,483,647, representing the average number of rows returned by the procedure for a given set of input parameters.

io-count

Specifies an integer value, in the range 0 through 2,147,483,647, representing the average number of disk accesses generated by the procedure for a given set of input parameters.

USER MODE

Specifies that the procedure should execute as a user-mode application program within CA IDMS. This can not be specified with language SQL or protocol ADS. For other languages and protocols, it is the default.

SYSTEM MODE

Specifies that the procedure should execute as a system mode application program. SYSTEM MODE is the default if language is SQL.

To execute as a system mode application, the program must be fully reentrant and be written in either:

- ADS as a mapless dialog
- Assembler using DC calling conventions
- COBOL or PL/I and compiled with an LE-compliant compiler
local-stge-size

Specifies an integer, in the range 0 through 32767, which represents the size, in bytes, of a local storage area which CA IDMS allocates at runtime and passes to the procedure on each invocation.

Note: If you do not code a LOCAL WORK AREA clause, the default local storage size is 1024 bytes.

global-stge-size

Specifies an integer, in the range 0 through 32767, representing the size, in bytes, of the global storage area that CA IDMS allocates at runtime and passes to the procedure on each invocation.

CA IDMS allocates a global storage area once within a transaction and retains it until the transaction terminates.

key-id

Specifies the one- to four-character identifier for the global storage area. CA IDMS passes the same piece of global storage within a transaction to all SQL routines that have the same global storage key.

If you do not specify the storage key, CA IDMS allocates each procedure its own global storage area, which is not used for any other routine within the transaction.

TRANSACTION SHARING

Specifies whether transaction sharing should be enabled for database sessions started by the procedure. If transaction sharing is enabled for a procedure’s database session, it shares the current transaction of the SQL session. If language SQL is specified, TRANSACTION SHARING must be ON or the clause must not be specified.

ON

Specifies that transaction sharing should be enabled. ON is the default if language is SQL.

OFF

Specifies that transaction sharing should be disabled.

DEFAULT

Specifies that the transaction sharing setting in effect when the procedure is invoked should be retained. Default is the default for languages other than SQL.
**compile-option**

Specifies a CA ADS option to be used when compiling the dialog associated with an SQL procedure. The options that can be specified and the syntax to use are given in the *CA ADS Reference Guide*, Appendix D.2.6 Dialog-expression. *Compile-option* can be specified only if language is SQL.

**Note:** The ability to specify the ADS COMPILE OPTION clause is a CA IDMS extension.

**procedure-statement**

Specifies the actions taken in the procedure. *Procedure-statement* is required if language is SQL. It cannot be specified otherwise.

Expanded syntax for *procedure-statement* is shown above immediately following the CREATE PROCEDURE syntax. Descriptions for these parameters are located at the end of this section.

**DEFAULT DATABASE**

Specifies whether a default database should be established for database sessions started by the procedure.

**NULL**

Specifies that no default database should be established.

**CURRENT**

Specifies that the database to which the SQL session is connected should become the default for any database session started by the procedure.

**timestamp-value**

Specifies the value of the synchronization stamp to be assigned to the procedure. *Timestamp-value* must be a valid external representation of a timestamp.

**DYNAMIC RESULT SETS**

Defines the maximum number of result sets that a procedure invocation can return to its caller. A result set is a sequence of rows specified by a *cursor-specification*, created by the opening of a cursor and ranged over that cursor.

**maximum-dynamic-result-sets**

Defines an integer in the range 0-32767 specifying the maximum number of result sets a procedure can return. The default is 0.
Parameters for Expansion of parameter-definition

**parameter-name**

Specifies a 1- to 32-character name of a parameter to be passed to the table procedure. *Parameter-name* must:

- Be unique within the table procedure that you are defining
- Follow the conventions for SQL identifiers

All parameters are implicitly nullable. Input parameters can be assigned NULL as a parameter value and output parameters can return NULL.

**data-type**

Defines the data type for the named parameter. For expanded *data-type* syntax, see Expansion of Data-type.

**WITH DEFAULT**

Directs CA IDMS to pass a default value for the named parameter if no value for the parameter is specified.

The default value for a parameter is based on its data type:

<table>
<thead>
<tr>
<th>Column data type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>Blanks</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>A character string literal with a length of zero (that is, '')</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>Double-byte blanks</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>A double-byte character string literal with a length of zero</td>
</tr>
<tr>
<td>DATE</td>
<td>The value in the CURRENT DATE special register</td>
</tr>
<tr>
<td>TIME</td>
<td>The value in the CURRENT TIME special register</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>The value in the CURRENT TIMESTAMP special register</td>
</tr>
<tr>
<td>All numeric data types</td>
<td>0 (zero)</td>
</tr>
</tbody>
</table>
CREATE PROCEDURE

Parameters for Expansion of language-clause

**ADS**
- Specifies that the SQL routine is written in the CA ADS language.

**ASSEMBLER**
- Specifies that the SQL routine is written in the assembler language.

**COBOL**
- Specifies that the SQL routine is written in the COBOL language.

**PLI**
- Specifies that the SQL routine is written in the PL/I language.

**SQL**
- Specifies that the SQL routine is written in the SQL language.

**Note:** The ability to specify ADS or ASSEMBLER as a language is a CA IDMS extension.

Parameters for Expansion of procedure-statement

**SQL-AM-mgmt-stmt**
- Specifies a statement from the Access Module Management Statements category.

**SQL-authorization-stmt**
- Specifies a statement from the Authorization Statements category.

**SQL-Control-stmt**
- Specifies a statement from the Control Statements category.

**SQL-Diagnostics-stmt**
- Specifies a statement from the Diagnostics Statements category.

**SQL-DDL-stmt**
- Specifies a statement from the Data Description Statements category.

**SQL-DML-stmt**
- Specifies a statement from the Data Manipulation Statements category.

**SQL-session-mgmt-stmt**
- Specifies a statement from the Session Management Statements category.

**Note:** The ability to include a RELEASE, SUSPEND, or RESUME statement in an SQL routine is a CA IDMS extension.

**SQL-transaction-mgmt-stmt**
- Specifies a statement from the Transaction Management Statements category.

**Note:** The ability to include a COMMIT or ROLLBACK statement in an SQL routine is a CA IDMS extension.
Usage

Influencing Join Strategies

CA IDMS uses estimated row and I/O counts in determining the cost of joining a procedure with other tables, views, procedures or table procedure. To determine the optimal access strategy, CA IDMS examines different sequences for retrieving information. By providing the estimated row and I/O counts for both the procedure and for each access key used by the procedure, CA IDMS can select the optimal access strategy.

In determining the cost of a specific access strategy, CA IDMS uses estimates provided in CREATE PROCEDURE unless input values are available for each of the parameters included in a key. If values are available for each of these parameters, CA IDMS uses the estimates specified in the CREATE KEY statement instead of those specified in CREATE PROCEDURE.

Specifying a Synchronization Stamp

When defining or altering a procedure, you can specify a value for its synchronization stamp. You should use care when doing so because the purpose of the stamp is to enable the detection of discrepancies between an entity and its definition. If explicitly specified, you must set the synchronization stamp to a new value following a change so that the change is detectable by the runtime system.

If not specified, the synchronization stamp is automatically set to the current date and time.

Coding procedures with language SQL

The rules for coding the procedure body of an SQL procedure are given by procedure-statement. A procedure body typically contains multiple SQL statements and according to the SQL grammar, SQL statements are terminated by the semi-colon. However, to define SQL routines, the Command Facility (OCF, IDMSBCF, or Visual DBA OCF console) needs to be used. It also has the semi-colon as the default command terminator. Before a new command can be specified, the CREATE PROCEDURE needs to be terminated by a semi-colon. Clearly, the semi-colon cannot concurrently be used as a terminator by both the SQL procedure language and the Command Facility. Therefore, when procedure-statement contains multiple SQL statements or when the ADS COMPILE OPTION is specified, the Command Facility needs to use a terminator different from the semi-colon. To accomplish this, a SET OPTIONS COMMAND DELIMITER 'delimeter-string' must be executed. Changing the terminator of the Command Facility remains in effect until the end of the session or until a new SET OPTIONS COMMAND DELIMITER is encountered. For more information about SET OPTIONS, see the Command Facility chapter in the CA IDMS Common Facilities Guide.
Language SQL

If LANGUAGE SQL is specified, the following attribute settings are established by default and must not be overridden to a different value:

- Protocol is ADS
- Mode is SYSTEM
- Transaction sharing is ON

Procedures whose language is SQL are implemented through an automatically generated CA ADS dialog whose name is `external-routine-name`.

An error while parsing `procedure-statement` or an error while compiling the associated CA ADS dialog causes the CREATE PROCEDURE statement to terminate with a warning instead of a statement error. This allows the erroneous `procedure-statement` syntax to be saved in the catalog for later correction using the DISPLAY PROCEDURE command. The CA ADS dialog and associated access module are not created.

Specifying CA ADS Compile Options

If LANGUAGE SQL is specified, you can specify one or more compile options to be used when the associated dialog is compiled. Specifying compile options can be useful for debugging purposes to enable tracing and the use of online debugging facilities. Compile options can also be used to include additional work records and SQL tables which can be referenced in native CA ADS code included in the routine body.

Some useful compile options include:

- SYMBOL TABLE IS YES - to allow the use of symbols by the TRACE command and the online debug facilities
- ADD RECORD record-name - to enable manipulation of elements from the specified record
- ADD SQL TABLE table-name - to enable manipulation of columns or parameters of the specified SQL table-like object

Grouping procedure statements into a single statement

Multiple procedure statements can be grouped together as a compound statement. A compound statement is a control statement and therefore is also a procedure statement.

Dynamic Result Sets

An SQL invoked procedure can return one or more result sets to its caller, up to the maximum number specified by its dynamic result sets attribute. A result set is returned for each returnable cursor that is still open when the procedure returns control to its caller.
Example

The following CREATE PROCEDURE statement defines a procedure.

```sql
create procedure emp.get_bonus
    (emp_id unsigned numeric(4)    with default,
     bonus unsigned numeric(10)   with default,
     currency_bonus char(3)                    with default)
    external name getbonus
    protocol idms;
```

The procedure USER01.TSELECT1 uses the given employee ID to retrieve the first and last name. It returns the edited name in the RESULT parameter.

```sql
create procedure USER01.TSELECT1
    ( TITLE       varchar(10) with default
    , P_EMP_ID    numeric(4)
    , RESULT      varchar(20)
    )
    EXTERNAL NAME TSELECT1 LANGUAGE SQL
    select trim(EMP_FNAME) || ' ' || trim(EMP_LNAME)
    into RESULT
    from DEMOEMPL.EMPLOYEE
    where EMP_ID = P_EMP_ID
;
```

call user01.tselect1('TSIGNAL3', 1003);

```
+ TITLE       P_EMP_ID  RESULT
+----------    --------  -------
+ TSIGNAL3        1003  Jim Baldwin
```
The GET_EMPLOYEE_INFO procedure uses the given employee ID, to construct two result set cursors:

- A static declared cursor RET_COVERAGE returns a cursor with the data from the COVERAGE table.
- The allocated dynamic cursor RET_BENEFITS to return the data from the BENEFITS data.

```
set options command delimiter '++';
create procedure SQLROUTE.GET_EMPLOYEE_INFO
  ( TITLE       varchar(10) with default
    , P_EMP_ID    numeric(4)
    , RESULT      varchar(20)
  )
  EXTERNAL NAME GETEMPIN LANGUAGE SQL
  DYNAMIC RESULT SETS 2
begin not atomic
  declare STMNT_NAME    char(10) default 'DYN_STMT1';
  declare STMNT_BUF     char(80) default ' ';
  declare RET_COVERAGE cursor with return for
    select * from DEMOEMPL.COVERAGE
    where EMP_ID = P_EMP_ID;
  open RET_COVERAGE;
  set STMNT_BUF = 'select * from DEMOEMPL.BENEFITS'
      || 'where EMP_ID = ' || P_EMP_ID;
  prepare STMT_NAME from STMT_BUF;
  allocate 'RET_BENEFITS' cursor with return for STMT_NAME;
  open 'RET_BENEFITS';
  set RESULT = '2 returned result sets';
end
set options command delimiter default ++
```
More Information

- For more information about expanded procedure references, see Expansion of Procedure-reference.
- For more information about coding external routines, see Defining and Using Procedures.
- For more information about Control Statements, see Control Statements.
- For more information about Diagnostics Statements, see GET DIAGNOSTICS.
- For more information about the other categories, see Statement Categories.
- For more information about defining a returnable cursor, see ALLOCATE CURSOR or DECLARE CURSOR.
- For more information about how the caller processes the returned result sets, see ALLOCATE CURSOR.
- For more information about CALL, see CALL.
- For more information about DESCRIBE CURSOR, see DESCRIBE CURSOR.
- For more information about CREATE KEY, see CREATE KEY.
- For more information about the SET OPTIONS COMMAND DELIMITER, see the "Using SET OPTIONS to Select Options" topic in the CA IDMS Common Facilities Guide.

CREATE SCHEMA

The CREATE SCHEMA data description statement defines a schema in the dictionary.

Authorization

To issue a CREATE SCHEMA statement, you must have the CREATE privilege on the schema named in the statement.

If you specify FOR NONSQL SCHEMA, you must have the USE privilege on the non-SQL schema.

If you specify DBNAME, you must have USE privilege on the database; if you do not specify DBNAME or specify a value of NULL, you must have DBADMIN privilege on DBNAME SYSTEM.
CREATE SCHEMA

Syntax

```
CREATE SCHEMA schema-name

DEFAULT AREA segment-name.area-name
FOR NONSQL SCHEMA nonsql-schema-specification
FOR SQL SCHEMA sql-schema-specification
```

Expansion of nonsql-schema-specification

```
dictionary-name. nonsql-schema-name VERSION version-number

DBNAME nonsql-database-name
```

Expansion of sql-schema-specification

```
sql-schema-name

DBNAME sql-database-name
```

Parameters

**schema-name**

Specifies the name of the schema being created. *Schema-name* must be a 1-through 18-character name that follows the conventions for SQL identifiers. *Schema-name* must be unique within the dictionary.

**DEFAULT AREA**

Specifies the default area for storing rows of tables associated with the named schema. This area is used for any such table that is not explicitly assigned an area in the CREATE TABLE statement.

**segment-name.area-name**

Identifies the segment and area.

You do not need to define the named segment or area in the dictionary before issuing the CREATE SCHEMA statement.

**nonsql-schema specification**

Identifies the nonSQL-defined schema to associate with the SQL schema.

Expanded syntax for **nonsql-schema-specification** appears immediately following the statement syntax. Descriptions for these parameters are located at the end of this section.

**sql-schema-specification**

Identifies an existing SQL-defined schema to which the new SQL schema refers. Expanded syntax for sql-schema-specification appears immediately following the statement syntax.
Parameters for Expansion of nonsql-schema-specification

*nonsql-schema-name*

Names the nonSQL-defined schema.

*dictionary-name*

Names the dictionary that contains the nonSQL-defined schema.

If you do not specify *dictionary-name*, it defaults to the dictionary to which the SQL session is connected.

**VERSION version-number**

Identifies the version number of the nonSQL-defined schema. If VERSION *version-number* is not specified, *version-number* defaults to 1.

**DBNAME nonsql-database-name**

Identifies the database containing the data described by the nonSQL-defined schema. *nonsql-database-name* must be a segment name or a database name that is defined in the database name table.

If you do not specify DBNAME, no database name is included in the definition of *schema-name*. At runtime the database to which the SQL session is connected must include segments containing the areas described by the non-SQL-defined schema.

For considerations about whether to specify the database when you create a schema for a non-SQL-defined schema, see "Usage," later in this section.

Parameters for Expansion of sql-schema-specification

*sql-schema-name*

Names the referenced SQL-defined-schema. This named schema must not itself reference another schema.

**DBNAME sql-database-name**

Identifies the database containing the data described by the referenced SQL-defined schema. *sql-database-name* must be a database name that is defined in the database name table or a segment name defined in the DMCL.

If you do not specify DBNAME, no database name is included in the definition of *schema-name*. At runtime, the database to which the SQL session is connected must include segments containing the areas described by the referenced SQL-defined schema.
Usage

If You Omit DEFAULT AREA

If you do not associate a default area with the schema, you must assign an area to each table that you associate with the schema in a CREATE TABLE statement. You use the IN parameter of CREATE TABLE to assign an area to a table.

Creating a Referencing Schema

If either a FOR NONSQL SCHEMA or a FOR SQL SCHEMA clause is specified, then the new SQL-defined schema that is being created is said to reference the specified schema and itself becomes a referencing schema. If a non-SQL-defined schema is specified, then creation of a referencing schema enables SQL access to a non-SQL-defined database described by the referenced schema. Similarly, if the referenced schema is SQL-defined, then the creation of a referencing schema enables SQL access to an SQL-defined database described by the referenced schema.

In either case, if a DBNAME is specified, the referencing schema provides access to the database instance identified by database-name. If no DBNAME is specified, the referencing schema is unbound and the instance of the database to be accessed is determined at runtime. Access modules that reference tables through an unbound referencing schema can therefore be used to access more than one instance of a database.

You cannot define either a table or a view in a referencing schema. However, you can define a view in another schema that references a table through a referencing schema.
CREATE SCHEMA

Chapter 8: Statements

Specifying non-SQL-DBNAME

When you create a schema for a non-SQL-defined schema, you use the DBNAME parameter to specify the name of the database containing the data. The name specified can be the name of a segment or a database name defined in the database name table.

If you do not specify a database name, the database to which your SQL session is connected when accessing the non-SQL-defined tables must include the segments containing the data.

**Note:** For more information about defining a schema for a non-SQL-defined schema, see SQL Schema Considerations.

Specifying SQL DBNAME

When you create a referencing schema, you use the DBNAME parameter to specify the name of the database containing the data. The name specified can be either the name of a database name defined in the database name table or the name of a segment included in the DMCL.

If you do not specify a database name, the database to which your SQL session is connected when accessing the data through the referencing schema must include the segments containing the data.

Examples

Defining a Schema with a Default Area

The following CREATE SCHEMA statement defines the schema SALES. The default area for the schema is SALES_SEG.SALES_AREA.

```sql
create schema sales
default area sales_seg.sales_area;
```

Defining a Schema for a Non-SQL-defined Schema

In this example, the statement creates schema SALES for a non-SQL schema:

```sql
create schema sales
for nonsql schema corpdict.sales version 100;
```

Defining a Schema for an SQL-defined Schema

The following CREATE SCHEMA statement defines a schema for an SQL-defined schema:

```sql
create schema any_sales for sql schema sales;
```
CREATE TABLE

More Information

■ For more information about defining schemas, see ALTER SCHEMA and DROP SCHEMA (see page 428).
■ For more information about non-SQL-defined schemas, see the CA IDMS Database Administration Guide.

CREATE TABLE

The CREATE TABLE data description statement defines a table in the dictionary. Tables defined with the CREATE TABLE statement are called base tables.

Authorization

To issue a CREATE TABLE statement, you must:

■ Own the schema where the table is being defined or hold the CREATE privilege on the named table
■ Hold the USE privilege on the area where rows of the named table are stored

Syntax

```
CREATE TABLE schema-name. table-identifier

( column-definition, CHECK ( search-condition ) )

IN segment-name.area-name

COMRESS USING BUILTIN data-characteristic-table-name

ESTIMATED ROWS estimated-row-count

TABLE ID table-id-number

NO DEFAULT INDEX

TIMESTAMP timestamp-value
```

Expansion of column-definition

```
column-name data-type

NOT NULL

WITH DEFAULT
```
Parameters

`table-identifier`

Specifies the name of the table being treated. `Table-identifier` must be a 1- through 18-character name that follows the conventions for SQL identifiers.

`Table-identifier` must be unique among the table, view, function, procedure and table procedure identifiers within the schema associated with the table.

`schema-name`

Specifies the schema to be associated with the table. `Schema-name` must identify a schema defined in the dictionary.

If you do not specify `schema-name`, it defaults to:

- The current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically
- The schema associated with the access module used at runtime, if the statement is embedded in an application program

`column-definition`

Defines a column to be included in the table.

Columns are included in the table in the order they are specified.

The list of column definitions together with the CHECK parameter (if specified) must be enclosed in parentheses. Multiple column definitions must be separated by commas.

Expanded syntax for `column-definition` is shown immediately following the CREATE TABLE syntax. Descriptions for these parameters are located at the end of this section.

`CHECK (search-condition)`

Specifies criteria to be used to restrict the data that can be stored in the table. CA IDMS stores a new row in the table only if the value of `search-condition` is true for the row.

For expanded `search-condition` syntax, see Expansion of Search-condition. Restrictions on the use of `search-condition` in the CHECK parameter are discussed in "Usage" later in this section.
IN

Specifies the area to be used for storing rows of the table.

If you do not associate an area with a table, CA IDMS:

- Uses the default area, if any, for the schema associated with the table
- Returns an error if the schema does not have a default area

The IN parameter is a CA IDMS extension of the SQL standard.

\textit{segment-name}

Identifies the segment associated with the named area.

\textit{area-name}

Identifies the area to be associated with the table. \textit{Area-name} must identify an area defined in the dictionary.

COMPRESS

Specifies that data in the table is to be compressed before being stored in the database.

The COMPRESS parameter is valid only if CA IDMS Presspack is installed at your site.

The COMPRESS parameter is a CA IDMS extension of the SQL standard.

\textbf{USING \textit{data-characteristic-table-name}}

Specifies the data characteristic table CA IDMS Presspack is to use to compress data in the table.

\textit{Data-characteristic-table} must identify a data characteristic table created by CA IDMS Presspack. If \textit{data-characteristic-table} is not specified, the default, BUILTIN, directs CA IDMS Presspack to use the data characteristic table supplied with the product.
**ESTIMATED ROWS estimated-row-count**

Indicates the number of rows expected to be stored for the table.
*Estimated-row-count* must be an integer that does not exceed 16,777,214.

CA IDMS uses the estimated row count when determining default index characteristics and estimating statistics.

The ESTIMATED ROWS parameter is a CA IDMS extension of the SQL standard.

**TABLE ID table-id-number**

Assigns a table ID value for the table being created. The table-id number must be in the range of 1024 through 4095.

**NO DEFAULT INDEX**

Indicates that the TABLE will have no initially assigned default index. The default index is an index sorted by DBKEY in ascending order in such a way that all TABLE rows can be accessed with the minimum number of I/Os.

**Note:** For more information about retaining or dropping the default index, see the Usage topic later in this section or the CA IDMS Database Design Guide.

**TIMESTAMP timestamp-value**

Specifies the value of the synchronization stamp to be assigned to the table. *Timestamp-value* must be a valid external representation of a timestamp.

**Parameters for Expansion of column-definition**

**column-name**

Specifies the name of a column to be included in the table being created. *Column-name* must be a 1- through 32-character name that follows the conventions for SQL identifiers.

*Column-name* must be unique within the table being defined.

**data-type**

Defines the data type for the named column. For expanded *data-type* syntax, see Expansion of Data-type.

**NOT NULL**

Indicates the column cannot contain null values.

If you do not specify NOT NULL, the column is defined to allow null values.

If you specify NOT NULL without WITH DEFAULT, an INSERT statement must specify a value for the column.
WITH DEFAULT

Directs CA IDMS to store the default value in the named column if no value for the column is specified when a row is inserted.

The default value for a column is based on its data type:

<table>
<thead>
<tr>
<th>Column data type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>Blanks</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>A character string literal with a length of zero (that is, '')</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>Double-byte blanks</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>A double-byte character string literal with a length of zero</td>
</tr>
<tr>
<td>DATE</td>
<td>The value in the CURRENT DATE special register</td>
</tr>
<tr>
<td>TIME</td>
<td>The value in the CURRENT TIME special register</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>The value in the CURRENT TIMESTAMP special register</td>
</tr>
<tr>
<td>All numeric data types</td>
<td>0 (zero)</td>
</tr>
</tbody>
</table>
Usage

Tables in the SYSTEM Schema

You cannot define a table in the SYSTEM schema.

Tables in System Areas

You cannot associate a table with a system area supplied with CA IDMS.

Maximum Row Length

When defining the columns in a table, you must ensure that the total number of bytes required for all columns in the table does not exceed the maximum allowed.

The total number of bytes allowed for all columns included in a table defined with the COMPRESS option is 32,760. If the table is defined without the COMPRESS option, the total number of bytes allowed for all columns is limited by the database page size and the size of the page reserve. The length of all columns must be less than or equal to \((\text{page-size} - \text{page-reserve} - 40)\).

The number of bytes used for each column is determined by the column data type. Columns that allow null values take one additional byte each.

Each linked clustered referential constraint where the table is the referencing table reduces the total number of bytes allowed for columns by 12. Each linked clustered referential constraint in which the table is the referenced table or linked indexed referential constraint where the table is the referencing or the referenced table reduces the total by 8 bytes.

A CALC key defined on a table also reduces the total number of bytes allowed for columns by 8.
Recommended Row Length

The absolute maximum row length for an uncompressed table is \((\text{page-size} - \text{page-reserve} - 40)\). The recommended maximum row length is 30% of the absolute maximum.

Restrictions on search-condition

In the CHECK parameter of a CREATE TABLE statement:

- **Search-condition** cannot include any host variables, routine parameters, local variables, aggregate or user-defined functions, EXISTS predicates, quantified predicates, or subqueries
- Each column reference in search condition must identify a column in the table being defined

Default Indexes

The default index for a table is stored in the same area as the table. CA IDMS uses the default index to cluster rows of the table when no other clustered index, CALC key, or clustered referential constraint is defined for the table.

For such a table, the default index improves processing efficiency. CA IDMS uses the default index instead of an area sweep to locate rows of the table for retrieval.
The ESTIMATED ROWS Parameter with Large Tables

To enable CA IDMS to choose optimal attributes for indexes on a large table, you should supply an estimated number of rows in the table definition or specify index block characteristics yourself.

If you do not specify ESTIMATED ROWS and if you do not update statistics after the table has been loaded, CA IDMS calculates index characteristics using an estimated row count of 1000.

Omitting NOT NULL and WITH DEFAULT

If you omit both NOT NULL nor WITH DEFAULT, the column is assigned a null value if no value is specified for the column on an INSERT statement.

Specifying a Synchronization Stamp

When defining or altering a table, you can specify a value for its synchronization stamp. You should use care when doing so because the purpose of the stamp is to enable the detection of discrepancies between an entity and its definition. If explicitly specified, you must set the synchronization stamp to a new value following a change so that the change is detectable by the runtime system.

If not specified, the synchronization stamp is automatically set to the current date and time.
Example

Defining a Base Table

The following CREATE TABLE statement defines the EMPLOYEE table in the DEMO_LIB schema. The table includes 16 columns. The CHECK parameter in the table definition restricts the values that can be stored in the STATUS column. Data in the table is stored in a compressed form in the EMP_SPACE area. The expected number of rows for the table is 350.

```
create table demo_lib.employee
    (emp_id           integer        not null,
     manager_id        integer,
     emp_fname         varchar(20)    not null,
     emp_lname         varchar(20)    not null,
     dept_id           integer        not null,
     proj_id           varchar(10),
     street            varchar(40)    not null,
     city              character(20)  not null,
     state             character(2)   not null,
     zip_code          character(9)   not null,
     phone             character(10),
     status            character(1),
     ss_number         integer        not null,
     start_date        date           not null,
     termination_date  date,
     birth_date        date,
     check (status in ('A', 'S', 'L', 'T'))
    )
in demoseg.emp_space
compress
estimated rows 350;
```

More Information

- For more information about defining tables, see ALTER TABLE and DROP TABLE.
- For more information about implementing indexes, see the CA IDMS Database Design Guide.
- For more information about compressing data, see the CA IDMS Presspack User Guide.
- For more information about differences between the CREATE TABLE statement in CA IDMS and the SQL standard CREATE TABLE statement.
CREATE TABLE PROCEDURE

The CREATE TABLE PROCEDURE data description statement stores the definition of a table procedure in the SQL catalog. You can refer to the table procedure in SQL SELECT, INSERT, UPDATE and DELETE statements just as you would a table. These references result in CA IDMS calls to the corresponding external routine. Although such routines can perform any action, you use them typically to manipulate data stored in some other organization (for example, in a non-SQL-defined database or in a set of VSAM files).

You use the formal parameters of a table procedure definition like the columns of a table during a procedure invocation. You can input values in and return them from the table procedure using column-like syntax.

The CREATE TABLE PROCEDURE statement is a CA IDMS extension of the SQL standard.

Authorization

To issue a CREATE TABLE PROCEDURE statement, you must own the schema in which the table procedure is being defined or hold the CREATE privilege on the named table procedure.

Syntax

```
CREATE TABLE PROCEDURE  
  table-procedure-identifier 
  (  
    parameter-definition     
    )  
  EXTERNAL NAME external-routine-name  
  
  ESTIMATED ROWS row-count  
  ESTIMATED IOS io-count  
  
  USER MODE 
  SYSTEM MODE  
  
  LOCAL WORK AREA  
    local-stge-size  
  
  GLOBAL WORK AREA  
    global-stge-size     
      KEY key-id  
  
  TRANSACTION SHARING  
    ON 
    OFF  
    DEFAULT  
  
  DEFAULT DATABASE  
    NULL  
    CURRENT  
  
  TIMESTAMP  
    timestamp-value  
```

Expansion of parameter-definition

```
  parameter-name  
    data-type    
      WITH DEFAULT  
```
Parameters

**table-procedure-identifier**

Specifies the 1- to 18-character name of the table procedure you are creating. *table-procedure-identifier* must:

- Be unique among the table, view, function, procedure and table procedure identifiers within the schema associated with the table procedure
- Follow conventions for SQL identifiers

**schema-name**

Specifies the schema name qualifier to be associated with the table procedure. *Schema-name* must identify a schema defined in the dictionary. If you do not specify a *schema-name*, it defaults to:

- The current schema associated with your SQL session, if the statement is specified through the Command Facility or executed dynamically
- The schema associated with the access module used at runtime, if the statement is embedded in an application program

**parameter-definition**

Defines a parameter to be associated with the table procedure. Parameters are passed to the table procedure in the order they are specified. The list of parameters must be enclosed in parentheses. Multiple parameter definitions must be separated by commas.

Expanded syntax for **parameter-definition** is shown immediately following the CREATE TABLE PROCEDURE syntax. Descriptions for these parameters are located at the end of this section.

**external-routine-name**

Specifies the one- to eight-character name of the program which is called to process references to the table procedure.

**row-count**

Specifies an integer value, in the range 0 through 2,147,483,647, representing the average number of rows returned by the table procedure for a given set of input parameters.

**io-count**

Specifies an integer value, in the range 0 through 2,147,483,647, representing the average number of disk accesses generated by the table procedure for a given set of input parameters.

**USER MODE**

Specifies the table procedure should execute as a user-mode application program within CA IDMS. This is the default value unless SYSTEM MODE is specified.
SYSTEM MODE

Specifies the table procedure should execute as a system mode application program. To execute in system mode, the program must be fully reentrant and be written in either:

- Assembler using DC calling conventions
- COBOL or PL/I and compiled with an LE-compliant compiler

**local-stge-size**

Specifies an integer, in the range 0 through 32767, which represents the size, in bytes, of a local storage area which is allocated by CA IDMS at runtime and passed to the table procedure on each invocation.

CA IDMS allocates a local storage area on the first call to a table procedure for each SQL statement within a transaction or for a set of SQL statements which are related through reference to the same cursor (OPEN, FETCH, CLOSE, positioned UPDATE, and DELETE statements are related through a cursor). The same local storage area is passed to the table procedure for all calls for one statement or related statements. When the SQL statement has completed execution or when the cursor is closed, the local work area is released.

**Note:** If you do not code a LOCAL WORK AREA clause, the default local storage size is 1024 bytes.

**global-stge-size**

Specifies an integer, in the range 0 through 32767, representing the size, in bytes, of the global storage area that CA IDMS allocates at runtime and passes to the table procedure on each invocation.

CA IDMS allocates a global storage area once within a transaction and retains it until the transaction terminates.

**key-id**

Specifies the one- to four-character identifier for the global storage area. CA IDMS passes the same piece of global storage within a transaction to all SQL routines that have the same global storage key.

If you do not specify a storage key, CA IDMS allocates each table procedure its own global storage area, which is not used for any other routine within the transaction.
TRANSACTION SHARING

Specifies whether transaction sharing should be enabled for database sessions started by the table procedure. If transaction sharing is enabled for a procedure's database session, it will share the current SQL session's transaction.

ON

Specifies that transaction sharing should be enabled

OFF

Specifies that transaction sharing should be disabled.

DEFAULT

Specifies that the transaction sharing setting that is in effect when the table procedure is invoked should be retained.

DEFAULT DATABASE

Specifies whether a default database should be established for database sessions started by the table procedure.

NULL

Specifies that no default database should be established.

CURRENT

Specifies that the database to which the SQL session is connected should become the default for any database session started by the table procedure.

timestamp-value

Specifies the value of the synchronization stamp to be assigned to the table procedure. Timestamp-value must be a valid external representation of a timestamp.

Parameters for Expansion of parameter-definition

parameter-name

Specifies a 1- to 32-character name of a parameter to be passed to the table procedure. Parameter-name must:

- Be unique within the table procedure that you are defining
- Follow the conventions for SQL identifiers

All parameters are implicitly nullable. Input parameters can be assigned NULL as a parameter value and output parameters can return NULL.

data-type

Defines the data type for the named parameter. For expanded data-type syntax, see Expansion of Data-type.
WITH DEFAULT

Directs CA IDMS to pass a default value for the named parameter if no value for the parameter is specified.

The default value for a parameter is based on its data type:

<table>
<thead>
<tr>
<th>Column data type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>Blanks</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>A character string literal with a length of zero (that is, '')</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>Double-byte blanks</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>A double-byte character string literal with a length of zero</td>
</tr>
<tr>
<td>DATE</td>
<td>The value in the CURRENT DATE special register</td>
</tr>
<tr>
<td>TIME</td>
<td>The value in the CURRENT TIME special register</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>The value in the CURRENT TIMESTAMP special register</td>
</tr>
<tr>
<td>All numeric data types</td>
<td>0 (zero)</td>
</tr>
</tbody>
</table>

Usage

Influencing Join Strategies

CA IDMS uses estimated row and I/O counts in determining the cost of joining a table procedure with other tables, views, or table procedures. To determine the optimal access strategy, CA IDMS examines different sequences for retrieving information. By providing the estimated row and I/O counts for the table procedure and for each access key used by the table procedure, CA IDMS can select the optimal access strategy.

In determining the cost of a specific access strategy, CA IDMS uses estimates provided in CREATE TABLE PROCEDURE unless input values are available for each of the parameters included in a key. If values are available for each of these parameters, CA IDMS uses the estimates specified in the CREATE KEY statement instead of those specified in CREATE TABLE PROCEDURE.

Specifying a Synchronization Stamp

When defining or altering a table procedure, you can specify a value for its synchronization stamp. You should use care when doing so because the purpose of the stamp is to enable the detection of discrepancies between an entity and its definition. If explicitly specified, you must set the synchronization stamp to a new value following a change so that the change is detectable by the runtime system.

If not specified, the synchronization stamp is automatically set to the current date and time.
Example

The following CREATE TABLE PROCEDURE statement defines a table procedure.

```sql
create table procedure emp.org
    (top_key          unsigned numeric(4),
     level             smallint,
     mgr_id            unsigned numeric(4),
     mgr_lname         char(25),
     emp_id            unsigned numeric (4),
     emp_lname         char(25),
     start_date        DATE,
     structure_code    char(2))
external name procorgu
local work area 800
global work area 600 KEY EMP
estimated rows 100
estimated ios 50;
```

More Information

- For more information about expanded table procedure references, see Expansion of Table-procedure-reference.
- For more information about coding the external routine, see Defining and Using Table Procedures.
- For more information about CREATE KEY, see CREATE KEY.

CREATE TEMPORARY TABLE

The CREATE TEMPORARY TABLE data description statement defines a temporary table. A temporary table exists for the duration of the transaction in which the table is created. When the transaction ends, CA IDMS deletes the definition of and the data associated with the temporary table.

The CREATE TEMPORARY TABLE statement is a CA IDMS extension of the SQL standard.

Authorization

None required.

Syntax

```sql
CREATE TEMPORARY TABLE table-identifier
    ( column-definition )
```

```sql
CREATE TEMPORARY TABLE
    ( column-definition )
```
**Expansion of column-definition**

```plaintext
  column-name  data-type
    NOT NULL
    WITH DEFAULT
```

**Parameters**

**table-identifier**

Specifies the name of the temporary table being created. *Table-identifier* must be a 1- through 18-character name that follows the conventions for SQL identifiers.

*Table-identifier* must be unique within the transaction in which the temporary table is defined. To prevent possible ambiguity, temporary table identifiers should differ from the identifiers of any base tables and views defined in the dictionary.

**column-definition**

Defines a column to be included in the temporary table.

Columns are included in the table in the order they are specified.

The list of column definitions must be enclosed in parentheses. Multiple column definitions must be separated by commas.

Expanded syntax for *column-definition* is shown immediately following the CREATE TEMPORARY TABLE syntax. Descriptions for these parameters are located at the end of this section.

**Parameters for Expansion of column-definition**

**column-name**

Specifies the name of a column to be included in the temporary table. *Column-name* must be a 1- through 32-character name that follows the conventions for SQL identifiers.

*Column-name* must be unique within the temporary table being defined.

**data-type**

Defines the data type for the named column. For expanded *data-type* syntax, see Expansion of Data-type.

**NOT NULL**

Indicates that the column cannot contain null values.

If you specify NOT NULL without WITH DEFAULT, an INSERT statement must specify a value for the column.

If you do not specify NOT NULL, the column is defined to allow null values.
WITH DEFAULT

Directs CA IDMS to store the default value for the named data type in the named column if no value for the column is specified when a new row is stored.

The default value for a column is based on its data type:

<table>
<thead>
<tr>
<th>Column data type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>Blanks</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>A character string literal with a length of zero (that is, '')</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>Double-byte blanks</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>A double-byte character string literal with a length of zero</td>
</tr>
<tr>
<td>DATE</td>
<td>The value in the CURRENT DATE special register</td>
</tr>
<tr>
<td>TIME</td>
<td>The value in the CURRENT TIME special register</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>The value in the CURRENT TIMESTAMP special register</td>
</tr>
<tr>
<td>All numeric data types</td>
<td>0 (zero)</td>
</tr>
</tbody>
</table>

Usage

Maximum Row Length

The total number of bytes allowed for all columns included in a temporary table is 32,767. The number of bytes used for each column is determined by the column data type. Columns that allow null values take one additional byte each.

Example

Defining a Temporary Table

The following CREATE TEMPORARY TABLE statement defines the temporary table TEMP_BUDGET with two columns:

```
create temporary table temp_budget
  (dept_id integer not null,
   all_expenses decimal(9,2));
```

CREATE VIEW

The CREATE VIEW data description statement defines a view in the dictionary.
Authorization

To issue a CREATE VIEW statement, you must own the schema where the view is being defined or hold the CREATE privilege on the named view.

Syntax

```sql
CREATE VIEW view-identifier
  [schema-name]
  [ ( view-column-name ) ]
  AS query-expression
  [ order-by-specification ]
  [ WITH CHECK OPTION ]
  [ TIMESTAMP timestamp-value ]
```

Expansion of order-by-specification

```sql
ORDER BY column-name
  [ table-name. alias. column-number result-name rowid-pseudo-column ]
  [ ASC | DESC ]
```
CREATE VIEW

Parameters

view-identifier

Specifies the name of the view being created. View-identifier must be a 1- through 18-character name that follows the conventions for SQL identifiers.

View-identifier must be unique among the table, view, procedure and table procedure identifiers within the schema associated with the view.

schema-name

Specifies the schema to be associated with the view. Schema-name must identify a schema defined in the dictionary.

If you do not specify schema-name, it defaults to:

- The current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically
- The schema associated with the access module used at runtime, if the statement is embedded in an application program

( view-column-name )

Assigns names to the columns to be included in the view. The number of column names must be the same as the number of columns in the result table represented by query-expression. The first column name is assigned to the first column in the result table, the second column name to the second result column, and so on.

Column-name must be a 1- through 32-character name that follows the conventions for SQL identifiers and must be unique within the view being defined.

The list of column names must be enclosed in parentheses. Multiple column names must be separated by commas.

If you do not specify any column names, CA IDMS assigns to the columns in the view the same names as those of the result table of query-expression.
AS query-expression

Defines the columns to be included in the view. The first column in the result table is the first column in the view, the second result column is the second column in the view, and so on.

Note: For more information about expanded query-expression syntax, see Expansion of Query-expression.

order-by-specification

Specifies a sort order for the rows in the result table defined by query-expression. Expanded syntax for order-by-specification is shown immediately following the CREATE VIEW syntax.

The use of the ORDER BY parameter in a CREATE VIEW statement is a CA IDMS extension of the SQL standard.

WITH CHECK OPTION

Specifies that any row inserted or updated through the view must satisfy the search condition of the WHERE clause in the query specification. This means you cannot add data through a view that the view would prevent you from retrieving.

TIMESTAMP timestamp-value

specifies the value of the synchronization stamp to be assigned to the view. Timestamp-value must be a valid external representation of a timestamp.

Parameters for Expansion of order-by-specification

ORDER BY

Sorts the rows in the result table defined by query-expression in ascending or descending order by the values in the specified columns. Rows are ordered first by the first column specified, then by the second column specified within the ordering established by the first column, then by the third column specified, and so on.

You can specify from 1 through 254 columns in the ORDER BY parameter. Multiple columns must be separated by commas.

column-name

Specifies a sort column by name. Column-name must identify a column in the result table of the query expression.

table-name

Specifies the table, view, procedure or table procedure that includes the named column. For expanded table-name syntax, see Identifying Entities in Schemas.

alias

Specifies the alias associated with the table, view, procedure or table procedure that includes the named column. Alias must be defined in the FROM parameter of the query specification that makes up the query expression.
**column-number**

Specifies a sort column by the position of the column in the result table defined by `query-expression`. The first result column is in position 1.

*Column-number* must be an integer in the range 1 through the number of columns in the result table.

**result-name**

Specifies the sort column by the result name specified in the AS parameter of `query-expression`.

**rowid-pseudo-column**

Specifies a sort column as a ROWID pseudo-column. See Expansion of rowid-pseudo-column.

**ASC**

Indicates that the values in the specified column are to be sorted in ascending order. ASC is the default when you specify neither ASC nor DESC.

**DESC**

Indicates that the values in the specified column are to be sorted in descending order.

**Usage**

**Views on SYSTEM Tables**

You can define a view on a table in the SYSTEM schema, but you cannot associate the view with the SYSTEM schema.

**Required Column Names**

You must include column names in a CREATE VIEW statement when any one of the following is true:

- Two or more of the result columns specified in the query-expression have the same name
- One or more of the value expressions representing the columns in the result table include a literal, an arithmetic operation (unary or binary), or an aggregate function
- A column in the result table has been assigned an alias through the AS parameter of the query specification

**Restriction on query-expression**

In a CREATE VIEW statement, `query-expression` cannot include:

- Host variables, local variables, or routine parameters
- References to temporary tables
Grouped Views

If the query-expression in a CREATE VIEW statement includes a GROUP BY or HAVING parameter that is not contained in a subquery, the view defined by the statement is a grouped view.

Updateable Views

For a view to be updateable:

- The query-expression must be updateable
- The view definition must not contain an ORDER BY clause

Result columns derived from a value expression other than a simple column reference cannot be updated or inserted through a view.

Using WITH CHECK OPTION

WITH CHECK OPTION has meaning only if the view is updateable and cannot be specified if the WHERE clause of the query expression contains a subquery.

When a view defined with WITH CHECK OPTION is referenced in the FROM clause of a second view definition, the check criterion of the original view is applied to data inserted or updated through the second view. If the second view is part of a third view definition, the check criterion of the original view is applied to data inserted or updated through the third view, and so on.

If a view defined without WITH CHECK OPTION is referenced in the FROM clause of a second view that has a WITH CHECK OPTION, the search conditions in the WHERE clause of both view definitions must be satisfied by an UPDATE or INSERT statement that references the second view. This principle holds true regardless of the number of levels of view references involved.

Once WITH CHECK OPTION is encountered in a view definition, all subordinate views referenced by that view are treated as if their definitions also contain WITH CHECK OPTION.
Use of * in a View Definition

Avoid the use of * in the query expression to denote all columns of a table named in the FROM parameter. If * is used and new columns are added to the table, the view becomes invalid; it must be dropped and recreated.

Altering the definition of an underlying table does not impact the view if you explicitly identify columns in the view definition.

Specifying a Synchronization Stamp

When defining a view, you can specify a value for its synchronization stamp. You should use care when doing so because the purpose of the stamp is to enable the detection of discrepancies between an entity and its definition. If explicitly specified, you must set the synchronization stamp to a new value following a change so that the change is detectable by the runtime system.

If not specified, the synchronization stamp is automatically set to the current date and time.

Note: For more information about dropping view definitions, see DROP VIEW.

Examples

Specifying Column Names in a View Definition

The following CREATE VIEW statement defines a view with three columns derived from two tables. The definition of the third column includes aggregate functions and a binary arithmetic operation. Therefore, the CREATE VIEW statement must specify names for all the columns in the view.

```
create view emp_vacation
  (emp_id, dept_id, vac_time)
as select e.emp_id, dept_id, sum(vac_accrued) - sum(vac_taken)
  from employee e, benefits b
  where e.emp_id = b.emp_id
  group by dept_id, e.emp_id;
```

Defining an Updateable View

The following CREATE VIEW statement defines an updateable view:

```
create view emp_home_info
  as select emp_id, emp_lname, emp_fname, street, city, state,
     zip_code, phone
  from employee;
```
DEALLOCATE PREPARE

The DEALLOCATE PREPARE statement destroys a dynamically-compiled statement and all other dynamically-compiled statements that directly or indirectly reference it. You can use this statement only in SQL embedded in an application program.

Authorization

None required.

Syntax

```
DEALLOCATE PREPARE statement-name
```

Parameters

`statement-name`

Identifies the statement to be destroyed. It must identify a statement previously created using a PREPARE statement.

Usage

Effect on Dependent Statements

Upon successful execution of a DEALLOCATE PREPARE statement, the following actions have taken place:

- The target statement is destroyed.
- If the target statement was a `cursor-specification`, all cursors that reference the target statement are destroyed. If the cursors were open at the time the DEALLOCATE PREPARE statement was executed, they are first closed.
- If any dynamically compiled positioned UPDATE or DELETE statements reference a cursor being destroyed, they too are destroyed.

Examples

Destroying a Prepared Statement

The following statement destroys the local statement named S1 and any cursors that reference the statement. It also destroys any statements that reference the cursors.

```
EXEC SQL
  DEALLOCATE PREPARE S1
END-EXEC
```
DECLARE CURSOR

The DECLARE CURSOR data manipulation statement defines a cursor for a specified result table. Use this statement only in SQL that is embedded in a program.

Authorization

To issue a DECLARE CURSOR statement that includes a cursor specification, you must own or have the SELECT privilege on each table, view, table procedure, and function explicitly named in the cursor specification. Authorization checking for cursors that reference a statement is done during execution of the corresponding PREPARE statement.

Additional authorization requirements apply to each view explicitly named in the cursor specification, to each view explicitly named in the definition of such a view, to each view explicitly named in the definition of those views, and so forth.

For any such view, the owner of the view must own or have the grantable SELECT privilege on each table, view, table procedure, and function explicitly named in the view definition.

Syntax

```
DECLARE static-cursor-name GLOBAL CURSOR
FOR cursor-specification static-statement-name
```

```
Expansion of cursor-declaration
```

```
DECLARE static-cursor-name GLOBAL CURSOR
FOR cursor-specification static-statement-name
```

```
```

```
```
Parameters

Parameters for Expansion of cursor-declaration

**static-cursor-name**

Assigns a name to the cursor. *Cursor-name* must be a 1- through 18-character name that follows the conventions for SQL identifiers.

**GLOBAL**

Specifies the cursor can be used by other application programs sharing the access module that contains the cursor definition.

The GLOBAL parameter is not valid for cursors associated with result tables defined by dynamically compiled SELECT statements.

The GLOBAL parameter is a CA IDMS extension of the SQL standard.

**WITH RETURN**

Defines the cursor as a returnable cursor. If a returnable cursor is declared in an SQL-invoked procedure and is in the open state when the procedure returns to its caller, a result set is returned to the caller.

**WITHOUT RETURN**

Specifies that the cursor is not a returnable cursor. This is the default.

**FOR**

Defines the result table associated with the cursor.

**cursor-specification**

Specifies the result table in the form of a cursor definition. For expanded *cursor-specification* syntax, see Expansion of Cursor-specification.

**static-statement-name**

Specifies the result table in the form of a dynamically compiled SELECT statement. *Statement-name* must identify a statement named in a PREPARE statement.

You cannot use a dynamically-compiled SELECT statement to define the result table associated with a global cursor using the DECLARE CURSOR statement. This can be achieved using an ALLOCATE CURSOR statement.
Usage

Uniqueness of Cursor Names

Each cursor name must be unique within an application program. Global cursor names must be unique within an access module.

Updateable Cursors

The cursor defined by a DECLARE CURSOR statement is updateable if the cursor specification, contained in the DECLARE CURSOR statement or represented by the static-statement-name is updateable.

Defining Returnable Cursors

While any cursor can be defined as a returnable cursor using WITH RETURN, it only makes sense to do so in programs that are invoked as SQL-invoked procedures and that are defined with a non-zero dynamic result set attribute.

The invoker must use theALLOCATE CURSOR statement to associate returned result sets with received cursors for further processing.

Note: For more information about how the caller processes returned result sets, see ALLOCATE CURSOR (see page 258) and CALL (see page 306).
Examples

Declaring a Global Cursor with a Specified Row Order

The following DECLARE CURSOR statement defines a global cursor for a result table containing information about all current employees and consultants. The rows in the table are ordered first by last name, then by department, and then by employee identifier.

```sql
EXEC SQL
 DECLARE ALL_EMP_CURSOR GLOBAL CURSOR
 FOR SELECT DEPT_ID, EMP_ID, 'EMPLOYEE', EMP_LNAME,
 EMP_FNAME, STREET, CITY, STATE, ZIP_CODE
 FROM EMPLOYEE
 WHERE STATUS IN ('A', 'L', 'S')
 UNION SELECT DEPT_ID, CON_ID, 'CONSULTANT', CON_LNAME,
 CON_FNAME, STREET, CITY, STATE, ZIP_CODE
 FROM CONSULTANT
 ORDER BY 4, 5, 1, 2
ENDEXEC
```

Naming an Updateable Column

The following DECLARE CURSOR statement defines a cursor for a result table containing fiscal year 1999 bonus information for each employee. The statement specifies that the BONUS_AMOUNT column of the BENEFITS table can be updated through the cursor.

```sql
EXEC SQL
 DECLARE BONUS_CURSOR CURSOR
 FOR SELECT EMP_ID, BONUS_AMOUNT
 FROM BENEFITS
 WHERE FISCAL_YEAR = '99'
 FOR UPDATE OF BONUS_AMOUNT
ENDEXEC
```
Associating a Cursor with a Dynamically Compiled SELECT Statement

The DECLARE CURSOR statement shown next, defines a cursor for the result table derived from a dynamically compiled SELECT statement named DYN_PROJ_SELECT. The application program must include a PREPARE statement for DYN_PROJ_SELECT.

EXEC SQL
DECLARE PROJECT_CURSOR CURSOR FOR DYN_PROJ_SELECT
END-EXEC

Defining Returnable Cursors

The following DECLARE CURSOR statement is specified in an SQL-invoked procedure written in SQL. The cursor RET_COVERAGE returns a result set consisting of the rows of the table DEMOEMPL.COVERAGE for which the column EMP_ID equals the value of the parameter P_EMP_ID. To effectively return the result set, the cursor must be left open on the return from the procedure.

declare RET_COVERAGE cursor with return for
    select * from DEMOEMPL.COVERAGE
    where EMP_ID = P_EMP_ID;

More Information

- For more information about manipulating cursors, see CLOSE, FETCH, and OPEN (see page 494).
- For more information about sharing cursors within an access module, see DECLARE EXTERNAL CURSOR.
- For more information about the dynamic compilation of SQL statements, see PREPARE or the CA IDMS SQL Programming Guide.
- For more information about using cursors in an application program, see the CA IDMS SQL Programming Guide.

DECLARE EXTERNAL CURSOR

The DECLARE EXTERNAL CURSOR data manipulation statement identifies an externally-defined global cursor to be used by the application program. You can use this statement only in SQL that is embedded in a program. The DECLARE EXTERNAL CURSOR statement is a CA IDMS extension of the SQL standard.

Authorization

None required.
Syntax

 DECLARE static-cursor-name EXTERNAL CURSOR

Parameter

 static-cursor-name

 Specifies the name of a global cursor to be used by the application program. Static-cursor-name must identify a cursor defined by a DECLARE CURSOR statement with the GLOBAL option in another application program that shares an access module with the program containing the DECLARE EXTERNAL CURSOR statement.

Usage

Sharing Cursors

For one program (program B) to use a cursor defined in another program (program A):

■ Program A must:
  – Include a DECLARE CURSOR statement that defines the cursor as a global cursor
  – Include an OPEN statement that opens the cursor
  – Open the cursor before program B attempts to use the cursor

■ Program B must include a DECLARE EXTERNAL CURSOR statement that names the cursor

■ The two programs must:
  – Use the same access module
  – Execute within the same transaction

Example

Identifying an Externally Defined Global Cursor

The following DECLARE EXTERNAL CURSOR statement identifies ALL_EMP_CURSOR as an externally defined global cursor that is used in the application program:

 EXEC SQL
   DECLARE ALL_EMP_CURSOR EXTERNAL CURSOR
 END-EXEC

More Information

■ For more information about defining global cursors, see DECLARE CURSOR
■ For more information about using cursors in an application program, see the CA IDMS SQL Programming Guide.
DELETE

The DELETE data manipulation statement deletes one or more rows from a table.

Authorization

To issue a DELETE statement, you must:

- Hold the DELETE privilege on or own the table, view, or table procedure named in the FROM parameter
- Hold the SELECT privilege on or own each table, view, and table procedure explicitly named in a subquery in the search condition in the WHERE parameter

Additional authorization requirements apply to:

- A view named in the FROM parameter, each view named in the FROM parameter of such a view, each view named in the FROM parameters of those views, and so forth.
  
  For any such view, the owner of the view must hold the grantable DELETE privilege on or own the table, view, or table procedure named in the FROM parameter of the view definition.
- Each view named in the FROM parameter of a subquery in the search condition, each view named in the FROM parameter of such a view, each view named in the FROM parameters of those views, and so forth.
  
  For any such view, the owner of the view must hold the grantable SELECT privilege on or own each table, view, and table procedure named in the FROM parameter of the view definition.

Syntax

```
DELETE FROM table-reference [ alias ]
WHERE search-condition
      CURRENT OF cursor-name
      dynamic-name-clause
```

Expansion of dynamic-name-clause

```
LOCAL | GLOBAL cursor-name
```
Parameters

**FROM table-reference**

Specifies the table, view, or table procedure from which rows are to be deleted. Table-reference must not specify a procedure or a joined table. If table-reference identifies a view:
- The view must be updateable
- The applicable rows are deleted from the table from which the view is derived

For expanded `table-reference` syntax, see Expansion of Table-reference.

**alias**

Defines a new name to be used to identify the table, view, or table procedure within the `DELETE` statement. *Alias* must be a 1- through 18-character name that follows the conventions for SQL identifiers.

**WHERE**

Restricts the rows to be deleted. If the `DELETE` statement does not include the `WHERE` parameter, CA IDMS deletes all rows from the specified table, view, or table procedure.

**search-condition**

Specifies criteria a row must meet to be deleted:
- When the value of `search-condition` is true, the row is deleted
- When the value of `search-condition` is false or unknown, the row is not deleted

For expanded `search-condition` syntax, see Expansion of Search-condition.

**CURRENT OF**

Specifies only the row that corresponds to the current row of the named cursor is to be deleted.

**cursor-name**

Identifies the cursor whose current row will be deleted. *Cursor-name* must identify an open cursor previously defined by a DECLARE CURSOR statement within the application program or by an ALLOCATE CURSOR statement executed within the same SQL transaction.

**Note:** This option may only be used in a `DELETE` statement embedded in an application program.

**dynamic-name-clause**

Identifies the cursor whose current row will be deleted.

**Note:** This option may only be used in a `DELETE` statement dynamically compiled using a PREPARE or EXECUTE IMMEDIATE statement.
Parameters for Expansion of dynamic-name-clause

LOCAL

Indicates the named cursor has a local scope and was defined using a DECLARE CURSOR statement or an ALLOCATE CURSOR statement. The default is LOCAL.

GLOBAL

Indicates the named cursor was created by an ALLOCATE CURSOR statement and is global in scope.

cursor-name

Specifies the name of the cursor as an identifier. Cursor-name must identify an open cursor previously defined by a DECLARE CURSOR statement within the application program or by an ALLOCATE CURSOR statement executed within the same SQL transaction.

Usage

Searched Deletes

A DELETE statement that include the WHERE search-condition parameter or does not include the WHERE parameter at all is called a searched delete. Searched deletes may be entered through the Command Facility, executed dynamically, or embedded within application programs.

Positioned Deletes

A DELETE statement that includes the WHERE CURRENT OF CURSOR parameter is called a positioned delete. The cursor identified in the positioned delete statement must be updateable. Positioned deletes are valid only from within an application program.

Dynamic Positioned Deletes

A dynamic positioned DELETE statement is one that references a dynamic cursor. Such a DELETE statement may be embedded within an application program or created dynamically using a PREPARE or EXECUTE IMMEDIATE statement.

A positioned DELETE statement embedded in an application program may reference a static cursor or a dynamic cursor. A positioned DELETE statement created dynamically using a PREPARE or EXECUTE IMMEDIATE statement can only reference a dynamic cursor.
Ambiguous Cursor References

When a dynamic positioned DELETE statement is being created by a PREPARE or EXECUTE IMMEDIATE statement, it is possible that CA IDMS may not be able to determine which cursor is being referenced. This occurs if the application program contains a DECLARE CURSOR statement that defines a cursor having the referenced name and the program has also executed an ALLOCATE cursor statement that creates a cursor with the same name and a local scope. Under these conditions, CA IDMS cannot determine which of the two cursors is being referenced. To avoid such problems, it is advisable to use different names for cursors that are declared from those that are allocated with a local scope.

Restrictions on table-reference

In a searched delete, the table, view, or table procedure named in the FROM parameter of the DELETE statement cannot also be named in the FROM parameter of any subquery included in the specified search condition or, in the case of a view, in any search condition used in the view definition. This means that you cannot delete data from a table from which you select in a subquery.

In a positioned delete, the table, view, or table procedure named in the FROM parameter of the DELETE statement must also be named in the FROM parameter of the query specification used in the definition of the named cursor.

Restriction for Tables in Referential Constraints

If the table referenced in a DELETE statement is the referenced table in a referential constraint, and the referencing table in the referential constraint includes one or more rows whose key-column values match those of a row to be deleted, CA IDMS returns an error and does not delete the row.

Cursor Position After a Positioned Delete

After a positioned delete, the position of the cursor named in the DELETE statement is before the row that immediately followed the deleted row. If the deleted row was the last row in the result table associated with the cursor, the position of the cursor is after the last row.

Transaction State for the DELETE Statement

CA IDMS processes a DELETE statement only when the transaction state is read write.

Deleting Through a View

If you specify a view in the FROM clause of a DELETE statement, the view must be updateable, and only rows that can be retrieved through the view can be deleted through the view.
Examples

Requesting a Searched Delete

The following DELETE statement deletes rows from the BENEFITS table for employees that have been terminated (status T):

```sql
delete from benefits
  where emp_id in
    (select emp_id
     from employee
     where status = 'T');
```

Requesting a Positioned Delete

The following DELETE statement deletes the row of the EST_COST table that corresponds to the current row of the EST_COST_CURSOR cursor:

```sql
EXEC SQL
  DELETE FROM EST_COST
    WHERE CURRENT OF EST_COST_CURSOR
END-EXEC
```

Deleting All Rows

The following DELETE statement deletes all rows from the PROPOSED_BUDGET table:

```sql
delete from proposed_budget;
```

A Positioned DELETE Referencing a DECLAREd Cursor

The following statement deletes the current row of the cursor C1. C1 may be a dynamic or static cursor, and it must have been defined using a DECLARE CURSOR statement:

```sql
EXEC SQL
  DELETE FROM EMPLOYEE WHERE CURRENT OF C1
END-EXEC
```

A Positioned DELETE Referencing an ALLOCATEd Cursor

The following statement deletes the current row of a cursor whose name is specified in the variable CNAME. The referenced cursor must have been defined using an ALLOCATE CURSOR statement:

```sql
EXEC SQL
  DELETE FROM EMPLOYEE WHERE CURRENT OF :CNAME
END-EXEC
```
A Dynamically-compiled Positioned DELETE Statement

The following statement deletes the current row of local cursor C1. C1 may have been defined using a DECLARE CURSOR statement or an ALLOCATE CURSOR statement. In either case, the cursor name in the DELETE statement is specified as an identifier rather than as a literal or host variable:

EXEC SQL
   EXECUTE IMMEDIATE
      'DELETE FROM EMPLOYEE WHERE CURRENT OF LOCAL C1'
END-EXEC

Note: The keyword LOCAL is unnecessary since it is the default. Regardless of whether it is specified, if two local cursors named C1 have been defined, one using a DECLARE CURSOR statement and one using an ALLOCATE CURSOR statement, the EXECUTE IMMEDIATE statement fails on an ambiguous cursor error.

More Information

■ For more information about updateable views, see CREATE VIEW.
■ For more information about defining and manipulating cursors, see CLOSE, DECLARE CURSOR, FETCH, and OPEN (see page 494).
■ For more information about updateable result tables, see DECLARE CURSOR

DESCRIBE

The DESCRIBE data compilation statement directs CA IDMS to return information about a dynamically-compiled SQL statement into an SQL descriptor area.

You can use this statement only in SQL that is embedded in a program.

Authorization

None required.

Syntax

```sql
DESCRIBE OUTPUT statement-name
USING sql DESCRIPTOR descriptor-area-name1
```
**Note:** If DESCRIBE OUTPUT is specified or implied, you may only specify the INPUT USING parameter; similarly, if DESCRIBE INPUT is specified, you may only specify the OUTPUT USING parameter.

**Note:** For compatibility with earlier releases, you can specify "INTO sql descriptor" in place of "USING sql DESCRIPTOR"; however, this is an extension to the SQL standard.

### Parameters

#### INPUT/OUTPUT

Specifies the type of information to be returned in the associated descriptor area. INPUT means that information about dynamic parameters is to be returned in the SQL descriptor area. OUTPUT means that information about output values is to be returned.

**statement-name**

Specifies the name of the statement being described.

**Note:** For more information about the expansion of **statement-name**, see Expansion of Statement-name.

#### USING SQL DESCRIPTOR

Specifies the SQL descriptor area where CA IDMS is to return information about the named statement.

**descriptor-area-name1**

Directs CA IDMS to use the named area as the descriptor area. *Descriptor-area-name1* must identify an SQL descriptor area.

#### INPUT/OUTPUT USING SQL DESCRIPTOR **descriptor-area-name2**

Specifies the type of information to be returned in the associated descriptor area. INPUT means that information about dynamic parameters is to be returned in the SQL descriptor area. OUTPUT means that information about output values is to be returned.

**Descriptor-area-name2** is the name of the SQL descriptor area.

**Note:** If DESCRIBE OUTPUT is specified or implied, you may only specify the INPUT USING parameter; similarly, if DESCRIBE INPUT is specified, you may only specify the OUTPUT USING parameter.

The ability to specify INPUT/OUTPUT USING SQL DESCRIPTOR **descriptor-area-name2** is a CA IDMS extension to the SQL standard.
Usage

Describing Dynamic Parameters

The INPUT option is used to return information about dynamic parameters that may be embedded in the SQL statement being described. The SQLD field of the descriptor area indicates the number of dynamic parameters that appear in the statement. If no dynamic parameters are used, this field is zero (0).

If dynamic parameters do appear in the statement, CA IDMS returns descriptions of the parameters in the descriptor area. The data type information is derived from the context in which the dynamic parameter appears.

Describing Output Values

The OUTPUT option is used to return information about values output from CA IDMS:

- For a SELECT statement, CA IDMS returns a description of the result table defined by the statement. The SQLD field of the descriptor area indicates the number of columns in the result table.
- For a statement other than SELECT, CA IDMS returns the value zero (0) in the SQLD field of the descriptor area.

Specifying the Maximum Number of Column Entries

The application program must specify the maximum number of entries it can accept by setting the value of the SQLN field of the descriptor area before issuing the DESCRIBE statement. If the number of entries is insufficient to hold all the requested information, CA IDMS returns the number of entries needed into the SQLD field but does not return any descriptions.

Example

Describing a Dynamically Compiled Statement

The following DESCRIBE statement returns information about the result table of the dynamically compiled statement named DYN_TEMP_SEL_1 in the descriptor area named SQLDA:

```
EXEC SQL
   DESCRIBE DYN_TEMP_SEL_1
   USING DESCRIPTOR SQLDA
END-EXEC
```

More Information

- For more information about the SQL descriptor area, see SQL Descriptor Area.
- For more information about the dynamic compilation of SQL statements, see the CA IDMS SQL Programming Guide.
DESCRIBE CURSOR

The DESCRIBE CURSOR data manipulation statement directs CA IDMS to return information about the result set associated with a received cursor into an SQL descriptor area.

Use this statement only in SQL that is embedded in a program.

Syntax

Parameters

extended-cursor-name

Specifies the name of the cursor whose result set is to be described. The cursor must have been previously associated with a returned result set using the ALLOCATE CURSOR statement.

USING sql DESCRIPTOR

Specifies the SQL descriptor area where CA IDMS is to return information about the result set with which the cursor is associated.

descriptor-area-name

Directs CA IDMS to use the named area as the descriptor area. descriptor-area-name must identify an SQL descriptor area.
Example

The GET_EMPLOYEE_INFO procedure returns two result sets for a given EMP_ID:
- One for COVERAGE info
- One for BENEFITS info

Note: For more information about how to define this procedure, see the examples in CREATE PROCEDURE.

* Invocation of the procedure GET_EMPLOYEE_INFO.

exec sql
    call GET_EMPLOYEE_INFO(1003)
end-

* The dynamic cursor 'RECEIVED_CURSOR' is associated with the first result set.

* The received cursor is in the open state.

exec sql
    allocate 'RECEIVED_CURSOR' for procedure specific procedure GET_EMPLOYEE_INFO
end-

* The 'RECEIVED_CURSOR' cursor info is being described

exec sql
    describe cursor 'RECEIVED_CURSOR' structure
        using sql descriptor SQLDA-AREA
end-

* The COVERAGE info is being processed.

* The statement is executed in a loop until the SQLSTATE indicates NO MORE DATA..

exec sql
    fetch 'RECEIVED_CURSOR' into :BUFFER-COVER
        using descriptor SQLDA-AREA
end-

* The dynamic cursor 'RECEIVED_CURSOR' is associated with the second result set.

* The received cursor is in the open state.
exec sql
close 'RECEIVED_CURSOR'
end-exec
.

* The 'RECEIVED_CURSOR' info is being described

exec sql
describe cursor 'RECEIVED_CURSOR' structure
                 using sql descriptor SQLDA-AREA
end-exec
.

* The BENEFITS info is being processed

* The statement is executed in a loop until the SQLSTATE indicates NO MORE DATA

exec sql
  fetch 'RECEIVED_CURSOR' into :BUFFER-BENEF
  using descriptor SQLDA-AREA
end-exec

* Close the cursor

exec sql
close 'RECEIVED_CURSOR'
end-exec

More Information

■ For more information about the SQL descriptor area, see SQL Descriptor Area.
■ For more information about calling procedures with dynamic results sets, see CALL.
■ For more information about allocating a cursor for a procedure, see ALLOCATE CURSOR.
■ For more information about closing a received cursor, see CLOSE.

DROP ACCESS MODULE

The DROP ACCESS MODULE access module management statement deletes an access module and its definition from the dictionary. It is also a CA IDMS extension of the SQL standard.

Authorization

To issue a DROP ACCESS MODULE statement, you must hold the DROP privilege on or own the access module named in the statement.
**Syntax**

```sql
drop access module  
  access-module-name  
  schema-name
  version  am-version-number
  preserve
```

**Parameters**

*access-module-name*

Specifies the name of the access module being dropped. *access-module-name* must identify an access module defined and stored in the dictionary.

*schema-name*

Identifies the schema associated with the specified version of the named access module.

If you do not specify *schema-name*, it defaults to the current schema associated with your SQL session.

*version  am-version-number*

Specifies the version number of the access module being dropped.

If VERSION is not specified, *am-version-number* defaults to 1.

*preserve*

Directs CA IDMS to retain privileges held on the access module being dropped. If you subsequently create a new access module with the same name as the access module being dropped, the preserved privileges will apply to the new access module.

If you do not specify PRESERVE in a DROP ACCESS MODULE statement, CA IDMS deletes all privileges held on the access module if CA IDMS internal security is in effect.

**Example**

**Dropping an Access Module**

The following DROP ACCESS MODULE statement deletes version 1 of the SALES001 access module from the dictionary but retains privileges held on the access module:

```sql
drop access module test.sales001
  preserve;
```
More Information

- For more information about access modules, see ALTER ACCESS MODULE and CREATE ACCESS MODULE (see page 320) or see the CA IDMS Database Administration Guide.
- For more information about CA IDMS internal security, see the CA IDMS Security Administration Guide.

DROP CALC

The DROP CALC data description statement deletes the definition of a CALC key from the dictionary. It is also a CA IDMS extension of the SQL standard.

Authorization

To issue a DROP CALC statement, you must own or have the ALTER privilege on the table on which the CALC key is defined.

Syntax

```
DROP CALC key FROM table-identifier
```

Parameters

**FROM table-identifier**

Specifies the name of the table associated with the CALC key being dropped. *Table-identifier* must identify a base table on which a CALC key has been defined in the dictionary. The named table cannot contain any data (that is, the table must be empty).

**schema-name**

Identifies the schema associated with the named table.

If you do not specify *schema-name*, it defaults to:

- The current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically
- The schema associated with the access module used at runtime, if the statement is embedded in an application program
Usage

**CALC Keys on Tables in the SYSTEM Schema**

You cannot delete the definition of a CALC key on a table in the SYSTEM schema.

**CALC Keys in the Implementation of Referential Constraints**

You cannot drop a CALC key that is used in the implementation of a referential constraint if no existing index can be used in place of the CALC key.

Example

**Dropping a CALC Key**

The following DROP CALC statement deletes the definition of the CALC key associated with the COVERAGE table from the dictionary:

```
drop calc from coverage;
```

More Information

- For more information about defining CALC keys, see CREATE CALC.
- For more information about unlinked referential constraints, see CREATE CONSTRAINT.

**DROP CONSTRAINT**

The DROP CONSTRAINT data description statement deletes the definition of a referential constraint from the dictionary. It is also a CA IDMS extension of the SQL standard.

Authorization

To issue a DROP CONSTRAINT statement, you must own or have the ALTER privilege on the referencing table in the constraint named in the statement.

Syntax

```
DROP CONSTRAINT constraint-name
FROM
  schema-name.table-identifier
```

Parameters

constraint-name

Specifies the name of the referential constraint being dropped. Constraint-name must identify a referential constraint defined in the dictionary.

table-identifier

Specifies the referencing table in the constraint to be dropped.

schema-name

Identifies the schema with which the table is associated.

If you do not specify a schema-name, the default value is:

■ The current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically
■ The schema associated with the access module used at runtime, if the statement is embedded in an application program

Usage

Referential Constraints Involving SYSTEM Tables

You cannot delete a constraint involving a table in the SYSTEM schema.

Implicitly Dropped Constraints

When you issue a DROP TABLE statement with the CASCADE parameter, CA IDMS deletes the definitions of all referential constraints in which the table being dropped is the referencing table or the referenced table.

Linked Constraints

When dropping a linked constraint, CA IDMS updates every row in the referenced and referencing tables to remove the physical links between the two tables.

Example

Dropping a Referential Constraint

The following DROP CONSTRAINT statement deletes the definition of the OFFICE_POOL_EMP constraint from the dictionary:

drop constraint office_pool_emp;
More Information

- For more information about defining referential constraints, see CREATE CONSTRAINT.
- For more information about implicitly dropping referential constraints, see DROP TABLE.

**DROP FUNCTION**

The DROP FUNCTION data description statement deletes the definition of the referenced function from the dictionary. For functions with language SQL, the statement removes the SQL routine body from the dictionary and the associated entities: access module (AM), relational command module (RCM), ADS premap process code, and dialog load module.

**Authorization**

To issue a DROP FUNCTION statement, you must own or have the DROP privilege on the function named in the statement.

**Syntax**

```
DROP FUNCTION [ schema-name. ] function-identifier [ CASCADE ]
```

**Parameters**

*function-identifier*

Specifies the name of the function to be dropped. Function-identifier must identify a function defined in the dictionary.

*schema-name*

Identifies the schema associated with the specified function.

If you do not specify a *schema-name*, the default value is:

- The current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically
- The schema associated with the access module used at runtime, if the statement is embedded in an application program

**CASCADE**

Directs CA IDMS to delete any view definition that contains a reference to the function, either directly or nested within some other view reference.
**Example**

The following DROP FUNCTION statement removes the FIN.UDF_FUNBONUS function from the SQL catalog:

```
DROP FUNCTION FIN.UDF_FUNBONUS CASCADE;
```

**More Information**

- For more information about syntax for creating functions, see CREATE FUNCTION.
- For more information about using functions, see Expansion of User-defined-function.
- For more information about coding the external routines which process function invocations, see Defining and Using Functions.

**DROP INDEX**

The DROP INDEX data description statement deletes the definition of an index from the dictionary. It is also a CA IDMS extension of the SQL standard.

**Authorization**

To issue a DROP INDEX statement, you must own or have the ALTER privilege on the table on which the index is defined.

**Syntax**

```
DROP INDEX index-name
FROM schema-name table-identifier
```
Parameters

*index-name*

Specifies the name of the index being dropped. *Index-name* must identify an index defined in the dictionary.

**FROM** *table-identifier*

Identifies the table on which the named index is defined.

*schema-name*

Identifies the schema associated with the named table.

If you do not specify a *schema-name*, the default value is:

- The current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically
- The schema associated with the access module used at runtime, if the statement is embedded in an application program

Usage

**Indexes on Tables in the SYSTEM Schema**

You cannot delete the definition of an index on a table in the SYSTEM schema.

**Indexes Used in the Implementation of Referential Constraints**

You cannot drop an index that is used in the implementation of a referential constraint if no other existing index or CALC key can be used in place of the index.

Example

**Dropping an Index**

The following DROP INDEX statement deletes the definition of the BUDGET_DATE_INDEX index from the dictionary:

```sql
drop index budget_date_index
from sales.monthly_budget;
```

More Information

- For more information about defining indexes, see CREATE INDEX.
- For more information about unlinked referential constraints, see CREATE CONSTRAINT.
DROP KEY

The DROP KEY statement deletes the definition of a procedure or table procedure key from the dictionary. It is also a CA IDMS extension of the SQL standard.

Authorization

To issue a DROP KEY statement, you must own or hold the ALTER privilege on the procedure or table procedure from which the key is being dropped.

Syntax

```
DROP KEY key-name
FROM schema-name. procedure-identifier
```

Parameters

- **key-name**
  Specifies the name of the key to be dropped. The *key-name* must identify a key defined in the dictionary.

- **FROM table-procedure-identifier**
  Specifies the table procedure from which the key is dropped.

- **procedure-identifier**
  Specifies the procedure from which the key drops.

- **schema-name**
  Identifies the schema associated with the named procedure or table procedure. If you do not specify a *schema-name*, it defaults to:
  - The current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically
  - The schema associated with the access module used at runtime, if the statement is embedded in an application program

Example

Dropping a Key

The following DROP KEY statement deletes the definition of the ORG1 key from the dictionary:

```
drop key org1 from emp.org;
```

**Note:** For more information about defining keys, see [CREATE KEY](page 357).
DROP PROCEDURE

The DROP PROCEDURE data description statement deletes the definition of the referenced procedure from the dictionary. For procedures with language SQL, the statement removes the SQL routine body from the dictionary and the associated entities: access module (AM), relational command module (RCM), ADS premap process code, and dialog load module.

Authorization

To issue a DROP PROCEDURE statement, you must own or have the DROP privilege on the procedure named in the statement.

Syntax

```
DROP PROCEDURE procedure-identifier
```

Parameters

**procedure-identifier**

Specifies the name of the procedure to be dropped. *Procedure-identifier* must identify a procedure defined in the dictionary.

**schema-name**

Identifies the schema associated with the specified procedure. If you do not specify a *schema-name*, the default value is:

- The current schema associated with your SQL session, if you specify the statement through the Command Facility or execute it dynamically
- The schema associated with the access module used at runtime, if the statement is embedded in an application program

**CASCADE**

Directs CA IDMS to delete any view definition that contains a reference to the procedure, either directly or nested within some other view reference.

Example

The following DROP PROCEDURE statement example removes the EMP.GET_BONUS procedure from the SQL catalog.

```
drop procedure emp.get_bonus cascade
```
DROP SCHEMA

More Information

- For more information about syntax for creating procedures, see CREATE PROCEDURE.
- For more information about defining and using procedures, see Defining and Using Procedures.
- For more information about coding the external routines which process procedure references, see Defining and Using Procedures.

DROP SCHEMA

The DROP SCHEMA data description statement deletes a schema definition from the dictionary.

Authorization

To issue a DROP SCHEMA statement, you must have the DROP privilege on the schema named in the statement.

Note: You need no additional privileges to issue a DROP SCHEMA statement with the CASCADE parameter.

Syntax

```
DROP SCHEMA schema-name [ CASCADE ]
```

Parameters

- **schema-name**
  
  Specifies the name of the schema being dropped. *Schema-name* must identify a schema defined in the dictionary.

- **CASCADE**
  
  Directs CA IDMS to perform a DROP TABLE CASCADE, DROP VIEW CASCADE, DROP PROCEDURE CASCADE, DROP TABLE PROCEDURE CASCADE, or a DROP FUNCTION CASCADE for each table, view, procedure, table procedure and function associated with the named schema.

  If you do not specify CASCADE in a DROP SCHEMA statement, the schema named in the statement cannot have any associated tables, views, functions, procedures and table procedures.
Usage

SYSTEM Schema

You cannot drop the SYSTEM schema.

Effect of the CASCADE Parameter

When you specify CASCADE in a DROP SCHEMA statement, CA IDMS deletes the following:

- The definition of each table, view, function, procedure and table procedure associated with the named schema
- The data stored in each table associated with the schema
- The definition of each referential constraint, index, and CALC key defined on the tables associated with the named schema
- The view definition of each view derived from one or more of the tables, views, functions, procedures or table procedures associated with the named schema
- For functions and procedures with language SQL, the statement removes the SQL routine body from the dictionary and the associated CA ADS entities and program structures: access module(AM), relational command module (RCM), ADS premap process code and dialog load module

Linked Constraints with Non-empty Tables in Other Schemas

If any tables in the schema to be dropped participate in linked referential constraints with non-empty tables in other schemas, CA IDMS also updates rows of those tables to remove the physical links with the tables being deleted.

Example

Dropping an Empty Schema

The following DROP SCHEMA statement deletes the definition of the SALES schema from the dictionary only if the schema has no associated tables, views, functions, procedures or table procedures:

drop schema sales;

Note: For more information about defining schemas, see ALTER SCHEMA and CREATE SCHEMA (see page 373).
The DROP TABLE data description statement deletes the definition of a base table from the dictionary.

When deleting a table definition, CA IDMS also deletes:
- The data contained in the table
- The CALC key, if any, defined on the table
- Any indexes defined on the table
- Optionally, referential constraints in which the table participates and views derived from the table

**Authorization**

To issue a DROP TABLE statement, you must own or have the DROP privilege on the table named in the statement.

**Note:** You need no additional privileges to issue a DROP TABLE statement with the CASCADE parameter.

**Syntax**

```
DROP TABLE [ schema-name. ] table-identifier [ CASCADE ]
```
Parameters

**table-identifier**

Specifies the name of the table being dropped. *Table-identifier* must identify a base table defined in the dictionary.

**schema-name**

Identifies the schema associated with the named table.

If you do not specify a *schema-name*, the default value is:

- The current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically
- The schema associated with the access module used at runtime, if the statement is embedded in an application program

**CASCADE**

Directs CA IDMS to delete the definitions of:

- All referential constraints where the named table is the referencing table or the referenced table
- All views derived from the named table

If you specify CASCADE in a DROP TABLE statement for a table that participates in a linked referential constraint, CA IDMS updates the rows of the other table to remove the physical links with the table being dropped.

Usage

**Tables in the SYSTEM Schema**

You cannot delete the definition of a table in the SYSTEM schema.

**Tables in Views or Referential Constraints**

If you do not specify CASCADE in a DROP TABLE statement, the table named in the statement cannot participate in the definition of any view or referential constraint.

Example

**Dropping a Table that Contains Data**

The following DROP TABLE statement deletes the definition of the OFFICE_POOL table and any data associated with the table. If the table participates in any referential constraint or view definitions, CA IDMS returns an error.

```
drop table office_pool;
```

**Note:** For more information about defining tables, see ALTER TABLE and CREATE TABLE (see page 378).
DROP TABLE PROCEDURE

The DROP TABLE PROCEDURE data description statement deletes the definition of the referenced table procedure from the dictionary. It is also a CA IDMS extension of the SQL standard.

Authorization

To issue a DROP TABLE PROCEDURE statement, you must own or have the DROP privilege on the table procedure named in the statement.

Syntax

```
DROP TABLE PROCEDURE schema-name. table-procedure-identifier [CASCADE]
```

Parameters

`table-procedure-identifier`

Specifies the name of the table procedure to be dropped. `Table-procedure-identifier` must identify a table procedure defined in the dictionary.

`schema-name`

Identifies the schema associated with the specified table procedure. If you do not specify a `schema-name`, the default value is:

- The current schema associated with your SQL session, if the statement is specified through the Command Facility or executed dynamically
- The schema associated with the access module used at runtime, if the statement is embedded in an application program

`CASCADE`

Directs CA IDMS to delete any view definition that contains a reference to the table procedure, either directly or nested within some other view reference.

Example

The following DROP TABLE PROCEDURE statement removes the EMP.ORG table procedure from the SQL catalog.

```
drop table procedure emp.org cascade
```
More Information

- For more information about syntax for creating table procedures, see CREATE TABLE PROCEDURE.
- For more information about defining and using table procedures, see Defining and Using Table Procedures.
- For more information about coding the external routines which process procedure references, see Defining and Using Table Procedures.

**DROP VIEW**

The DROP VIEW data description statement deletes the definition of a view from the dictionary.

**Authorization**

To issue a DROP VIEW statement, you must own or have the DROP privilege on the view named in the statement.

**Note:** You need no additional privileges to issue a DROP VIEW statement with the CASCADE parameter.

**Syntax**

\[
\text{DROP VIEW} \quad \text{schema-name.} \quad \text{view-identifier} \quad \text{CASCADE}
\]

**Parameters**

- **view-identifier**
  
  Specifies the name of the view being dropped. *View-identifier* must identify a view defined in the dictionary.

- **schema-name**
  
  Identifies the schema associated with the named view.

  If you do not specify a *schema-name*, the default value is:

  - The current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically
  - The schema associated with the access module used at runtime, if the statement is embedded in an application program

- **CASCADE**
  
  Directs CA IDMS to delete the definitions of all views derived from the named view.
Example

Dropping a View

The following DROP VIEW statement deletes the definitions of the EMP_VACATION view and all views derived from the EMP_VACATION view:

```
drop view emp_vacation cascade;
```

Views that Participate in Other Views

If you do not specify CASCADE in a DROP VIEW statement, the view named in the statement cannot participate in the definition of any other view.

Note: For more information about defining a view, see CREATE VIEW

END DECLARE SECTION

The END DECLARE SECTION statement is a precompiler directive that notifies the precompiler the SQL declare section has ended. You can use this statement only in SQL that is embedded in a program.

Authorization

None required.

Syntax

```
END DECLARE SECTION
```

Parameter

END DECLARE SECTION.

Notifies the precompiler that the SQL declare section has ended. An SQL declare section contains the definition of one or more host variables.
Example

Beginning and Ending an SQL Declaration Section

In this example, BEGIN DECLARE SECTION begins an SQL declare section and END DECLARE SECTION ends it. The SQL declare section contains the definition of five host variables.

WORKING-STORAGE SECTION.
.
.
EXEC SQL BEGIN DECLARE SECTION END-EXEC
  01 HV-EMP-ID    PIC S9(8) USAGE COMP.
  01 HV-EMP-LNAME PIC X(20).
  01 HV-SALARY-AMOUNT PIC S9(6)V(2) USAGE COMP.3.
  01 HV-PROMO-DATE PIC X(10).
  01 HV-PROMO-DATE-I PIC S9(4) USAGE COMP.
EXEC SQL END DECLARE SECTION END-EXEC

More Information

- For more information about beginning an SQL declare section, see BEGIN DECLARE SECTION.
- For more information about declaring host variables, see Host Variables or the CA IDMS SQL Programming Guide.

EXECUTE

The EXECUTE statement executes a dynamically-compiled SQL statement other than SELECT. You can use this statement only in SQL that is embedded in a program.

Authorization

None required.

Syntax
Expansion of bulk-options

- **START**: `start-variable-name`
- **ROWS**: `row-count-variable-name`

Expansion of dynamic-bulk-options1

- **START**: `start-variable-name`
- **ROWS**: `row-count-variable-name`
- **sql DESCRIPTOR**: `descriptor-area-name`

**Parameters**

**statement-name**

Identifies the statement being executed.

For detailed information, see Expansion of Statement-name.

**USING**

Supplies values for the dynamic parameters embedded in the text of the statement.

**host-variable**

Identifies the host variables from which CA IDMS is to retrieve values for the dynamic parameters. CA IDMS assigns the value of the first host variable to the first dynamic parameter, the second host variable to the second dynamic parameter, and so on.

You must specify the same number of host variables in the USING parameter as the number of dynamic parameter markers in the statement text.

*Note:* In COBOL, **host-variable** can be an elementary data item or a non-bulk structure. If a non-bulk structure is specified, each sub-element of the structure is counted as a host variable.

*Note:* For detailed information, see Expansion of Host-variable (see page 79).

**local-variable**

**routine-parameter**

Identifies the local variable or routine parameter from which CA IDMS is to retrieve values for the dynamic parameters. CA IDMS assigns the value of the first local variable or routine parameter to the first dynamic parameter, the second local variable or routine parameter to the second dynamic parameter, and so on. You must specify the same number of local variables and routine parameters in the USING parameter as the number of dynamic parameter markers in the statement text.
**:dyn-buff**

Identifies the variable or bulk-buffer from which CA IDMS is to retrieve values for the dynamic parameters.

_Dyn-buff_ must identify a variable previously declared in the host-language application program or SQL routine.

The size of _dyn-buff_ must be sufficient to hold a complete set of dynamic parameter values for a single execution of the statement. If specified as part of the BULK parameter, _dyn-buff_ must be sufficient to hold _row-count-variable_ sets of dynamic parameters. The format of the data in _dyn-buff_ must conform to the description in the SQL descriptor area specified by _descriptor-area-name_.

**BULK**

Directs CA IDMS to execute the statement one or more times and to use a contiguous storage area to retrieve input values for the dynamic parameters. The specification of BULK is a CA IDMS extension of the SQL Standard.

**Note**: BULK may only be specified if the statement being executed is an INSERT statement.

**:bulk-buffer**

Identifies a variable from which CA IDMS is to retrieve one or more sets of input values. _Bulk-buffer_ must identify a variable previously declared in the host-language application program or SQL routine.

_Bulk-buffer_ must be defined as a multiple-occurring structure having the same number of sub-elements as there are dynamic parameters in the statement.

**bulk-options**

Optionally specify the location in _bulk-buffer_ for the first row and the number of rows to be inserted. Expanded syntax for _bulk-options_ immediately follows the statement syntax.

**dynamic-bulk-options1**

Provides specification for inserting one or more rows into a table.

Expanded syntax for _dynamic-bulk-options1_ appears immediately following the expanded syntax for _bulk-options_. Descriptions of _dynamic-bulk-options1_ parameters appear above.
Note: *dyn-buff, bulk-buffer, start-variable-name, and row-count-variable-name* are variables that can be host variables or when the statement is used in an SQL routine local variables or routine parameters. In this case, their names must not be preceded with a colon.

Parameters for Expansion of bulk-options

**START :**<br>
*start-variable-name*

Identifies a variable containing the relative position within the bulk buffer from which CA IDMS is to retrieve values for the first row to be inserted. Values in subsequent entries in the bulk buffer are retrieved sequentially, each set corresponding to a row to be inserted.

*Start-variable-name* must be a variable previously declared in the host-language application program or SQL routine. The value in the variable must be an integer in the natural range of subscripts for arrays in the language in which the application program is written.

If you do not specify the START parameter, CA IDMS retrieves the values from the first entry in the bulk buffer.

**ROWS :**<br>
*row-count-variable-name*

Identifies a variable that specifies the number of rows CA IDMS is to retrieve from the bulk buffer.

*Row-count-variable-name* must be a variable previously declared in the host-language application program or SQL routine. The value in the variable must be in the range 1 through the number of rows that will fit in the bulk buffer.

If you do not specify the ROWS parameter, CA IDMS retrieves rows from the array sequentially until reaching the end of the buffer.

Parameters for Expansion of dynamic-bulk-options1

An additional parameter is used with *dynamic-bulk-options1*.

**SQL DESCRIPTOR**

Specifies the SQL descriptor area that describes the format of the dynamic parameter values contained in *dyn-buff*.

*descriptor-area-name*

Directs CA IDMS to use the named area as the descriptor area. *Descriptor-area-name* must identify an SQL descriptor area.
Usage

Dynamically-compiled SELECT Statements

You cannot use the EXECUTE statement with a dynamically-compiled SELECT statement. To retrieve data using a dynamically-compiled SELECT statement, you must define a cursor and use the FETCH statement.

Use of the Descriptor Area

When describing the format of dynamic parameters with an SQL descriptor area, you can use the INPUT option of the DESCRIBE statement to determine the format of the parameters that CA IDMS has assumed based on the context in which they appear. You can alter the contents of the descriptor area provided that the data types remain compatible. However, all changes to the descriptor area must be made before the first time the EXECUTE statement for the given dynamically-compiled statement is executed. The contents of the descriptor area must remain unchanged for each subsequent execution.

Examples

Executing a Dynamically-compiled Statement

The following EXECUTE statement executes the dynamically-compiled statement named DYN_PROJ:

```sql
EXEC SQL
   EXECUTE DYN_PROJ
END EXEC
```

More Information

- For more information about the dynamic compilation of SQL statements, see the CA IDMS SQL Programming Guide.
- For more information about the layout of an SQL descriptor area, see SQL Descriptor Area.

EXECUTE IMMEDIATE

The EXECUTE IMMEDIATE statement dynamically compiles and executes an SQL statement. You can use this statement only in SQL that is embedded in a program.

Authorization

To issue an EXECUTE IMMEDIATE statement, you must have the privileges required to issue the statement being dynamically compiled and executed.
EXECUTE IMMEDIATE

Syntax

EXECUTE IMMEDIATE 'sql-statement' :sql-statement-variable-name

Parameters

'sql-statement'

Specifies an SQL statement that can be compiled and executed immediately. Sql-statement must be enclosed in single quotation marks. Do not include the SQL prefix or terminator within the statement text.

:sql-statement-variable-name

Identifies a host variable, local variable, or routine parameter containing the statement to be compiled and executed immediately. Sql-statement-variable-name must be a variable previously declared in the application program or SQL routine. It must be defined as an elementary data item with no sub-elements. If sql-statement-variable-name is a local variable or routine parameter, the colon must not be coded.

Usage

Statements Eligible for Immediate Execution

The following SQL statements can be compiled and executed immediately:

- All access module management, authorization, logical data description, session management, and transaction management statements
- DELETE
- INSERT
- UPDATE

Additionally, all CA IDMS utility and physical data description statements can be compiled and executed immediately.

No Host Variables, Local Variables, or Routine Parameters in a Dynamically Compiled Statement

An SQL statement that is to be compiled dynamically cannot include any host variables, local variables, or routine parameters.

No Dynamic Parameters

An SQL statement that is compiled using the EXECUTE IMMEDIATE statement cannot include any dynamic parameter markers.

Note: For more information about the dynamic compilation of SQL statements, see the CA IDMS SQL Programming Guide.
Example

Using a Variable in EXECUTE

The following EXECUTE IMMEDIATE statement directs CA IDMS to dynamically compile and execute the statement contained in the variable DYN-INSERT:

EXEC SQL
   EXECUTE IMMEDIATE :DYN-INSERT
END-EXEC

EXPLAIN

The EXPLAIN utility statement describes the strategy used to access data in the following statements:

- DECLARE CURSOR
- DELETE
- SELECT
- UPDATE
- INSERT that contains a query specification in its VALUES clause

The description is stored as rows in a table which you can retrieve using a SELECT statement.

The EXPLAIN statement is a CA IDMS extension of the SQL standard.

Authorization

To issue an EXPLAIN statement that specifies:

- An access module, you must own or have the DISPLAY privilege on the access module being explained
- A statement, you must have the privileges required to execute the statement to be explained

Additionally, if the table in which the access plan is to be stored is:

- Already defined in the dictionary, you must own or have the INSERT privilege on the table
- Not already defined in the dictionary, you must:
  - Own the schema associated with the table or have the CREATE and INSERT privileges on the table
  - Have the USE privilege on the area in which rows of the table is stored
Syntax

```
EXPLAIN

access-module-specification
  STATEMENT 'sql-statement'
  STATEMENT NUMBER statement-number

  INTO TABLE schema-name table-identifier
  IN segment-name.area-name

Expansion of access-module-specification

  ACCESS MODULE access-module-name
  VERSION am-version-number
  MODULE rcm-name
```
Parameters

**access-module-specification**

Identifies an access module to be explained. Expanded syntax for **access-module-specification** is presented immediately following the EXPLAIN syntax.

**STATEMENT 'sql-statement'**

Directs CA IDMS to return the access strategy for the specified SQL statement. **Sql-statement** must be an explainable statement and must be enclosed in single quotation marks.

**STATEMENT NUMBER statement-number**

Assigns a reference number to the access plan for the statement specified in the STATEMENT parameter. The reference number is stored in the SECTION column in each row of the access plan.

**Statement-number** must be an integer in the range 0 through 32,767. If not specified, a value of 0 is returned.

**INTO TABLE table-identifier**

Specifies the table in which CA IDMS is to store the access plan. If you do not include the INTO TABLE parameter in an EXPLAIN statement, **table-identifier** is 'ACCESS_PLAN'.

If **table-identifier** does not exist, CA IDMS automatically defines it in the dictionary using the column definitions described in "Usage" following these parameter descriptions. If **table-identifier** identifies an existing table, the table must be defined with the appropriate columns for storing the access plan.

**Important!** Do not specify "EXPLAIN" as the **schema-name** or **table-identifier** where you will store the access plan. This produces an error message. The syntax parser interprets this as an attempt to perform a second EXPLAIN.

**schema-name**

Identifies the schema associated with the named table.

If you do not specify **schema-name**, it defaults to:

- The current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically
- The schema associated with the access module used at runtime, if the statement is embedded in an application program
IN `segment-name.area-name`

Identifies the area to be used for storing rows of the table named in the INTO TABLE parameter.

IN parameter information is used only when the INTO TABLE parameter identifies a table that does not exist.

If you do not specify the IN parameter, CA IDMS:

- Uses the default area, if any, for the schema associated with the table named in the INTO TABLE parameter
- Returns an error if the schema does not have a default area

Parameters for Expansion of access-module-specification

ACCESS MODULE `access-module-name`

Directs CA IDMS to describe the access strategy for all the explainable statements in the whole access module or in one or more specified RCMs in the access module.

`Access-module-name` must identify an access module stored in the DDLCATLOD area of the dictionary. Access modules in this area are represented in the SYSTEM.LOADHDR table.

**Note:** For more information about the SYSTEM.LOADHDR table, see SYSTEM.LOADHDR.

VERSION `am-version-number`

Specifies the version of the access module being explained.

If `am-version-number` is not specified, the version is 1.

MODULE `rcm-name`

Specifies one or more RCMs to be explained. CA IDMS will describe the access strategy for each explainable statement in each named RCM.

The SECTION value for the first explainable statement in the RCM is 0. The SECTION value for each succeeding explainable statement in the RCM is incremented by 1.

`rcm-name` must identify an RCM included in the access module named in the ACCESS MODULE parameter. Multiple RCM names must be separated by commas.

If you do not specify the MODULE parameter with ACCESS MODULE, CA IDMS explains all the RCMs in the named access module.
### Usage

**Explainable Statements**

The explainable statements are DECLARE CURSOR, DELETE, INSERT, SELECT, and UPDATE.

**Table ACCESS_PLAN**

The columns of the ACCESS_PLAN table are:

<table>
<thead>
<tr>
<th>Column</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBNAME</td>
<td>CHAR(8)</td>
<td>Dictionary connection for the session in which EXPLAIN is issued</td>
</tr>
<tr>
<td>ESTAMP</td>
<td>TIMESTAMP</td>
<td>Date and time EXPLAIN was issued</td>
</tr>
<tr>
<td>SCHEMA</td>
<td>CHAR(18)</td>
<td>Access module schema or, if explaining a statement, current schema for the SQL session</td>
</tr>
<tr>
<td>MODULE</td>
<td>CHAR(8)</td>
<td>Access module name or, if explaining a statement, IDMSEXPL</td>
</tr>
<tr>
<td>VERSION</td>
<td>SMALLINT</td>
<td>Access module version or, if explaining a statement, 0</td>
</tr>
<tr>
<td>STAMP</td>
<td>TIMESTAMP</td>
<td>Date and time access module was created, or, if explaining a statement, the same value as ESTAMP</td>
</tr>
<tr>
<td>PROGRAM</td>
<td>CHAR(8)</td>
<td>Program (RCM) name or, if explaining a statement, IDMSEXPL</td>
</tr>
<tr>
<td>PVERSION</td>
<td>SMALLINT</td>
<td>Program (RCM) version or, if explaining a statement, 0 (if explaining an access module, a version number of 0 indicates that no RCM version was specified the RCM when included in the access module)</td>
</tr>
<tr>
<td>PDICT</td>
<td>CHAR(8)</td>
<td>Program (RCM) dictionary or, if explaining a statement, blanks</td>
</tr>
<tr>
<td>PSTAMP</td>
<td>CHAR(20)</td>
<td>Date and time the program (RCM) was created or, if explaining a statement, blanks</td>
</tr>
<tr>
<td>SECTION</td>
<td>SMALLINT</td>
<td>Section number assigned to the SQLCSID field during program precompilation, or statement-number specified in the EXPLAIN statement</td>
</tr>
<tr>
<td>Column</td>
<td>Data type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>COMMAND</td>
<td>SMALLINT</td>
<td>Internal command code indicating the type of statement being explained:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8—DECLARE CURSOR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9—DELETE (searched)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10—DELETE (positioned)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17—INSERT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>25—SELECT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>29—UPDATE (searched)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30—UPDATE (positioned)</td>
</tr>
<tr>
<td>QBLOCK</td>
<td>SMALLINT</td>
<td>Query block number. Each query that the statement contains is assigned a block. Blocks are numbered beginning with 1.</td>
</tr>
<tr>
<td>STEP</td>
<td>SMALLINT</td>
<td>Step number. This number denotes the sequence of the processing step within the query block.</td>
</tr>
<tr>
<td>STYPE</td>
<td>SMALLINT</td>
<td>Step type. This denotes the type of processing for the step:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—Null</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—Table access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2—Nested loop join</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3—Merge join</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4—Sort</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5—Merge group</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6—OR list</td>
</tr>
<tr>
<td>PBLOCK</td>
<td>SMALLINT</td>
<td>Parent block number. Parent block numbers indicate nesting of multiple query blocks in a section.</td>
</tr>
<tr>
<td>PSTEP</td>
<td>SMALLINT</td>
<td>Parent step number. Parent step numbers correlate rows of query blocks:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ If a table scan row is owned by a sort or join row, PSTEP is the step number of the owning row.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ PSTEP of the top row of each main query block is 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ PSTEP of the top row of each subquery is the query block number of the main query block to which it is subordinate.</td>
</tr>
<tr>
<td>Column</td>
<td>Data type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TSCHEMA</td>
<td>CHAR(18)</td>
<td>Schema-name qualifier of the accessed table or procedure.</td>
</tr>
<tr>
<td>TABLE</td>
<td>CHAR(18)</td>
<td>Name of the accessed table or procedure.</td>
</tr>
<tr>
<td>TSTAMP</td>
<td>TIMESTAMP</td>
<td>Date and time the accessed table or procedure was created or last altered,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or the date and time the EXPLAIN was issued in case no table or procedure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>was accessed.</td>
</tr>
<tr>
<td>ACMODE</td>
<td>CHAR(1)</td>
<td>Mode of access to the database record underlying the table, when STYPE is 1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'A'—Area</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'C'—CALC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'I'—Index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'M'—Set member</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'N'—Insert</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'O'—Set owner</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'P'—Table procedure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'S'—Sequential</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'T'—(Temporary table)</td>
</tr>
<tr>
<td>ACNAME</td>
<td>CHAR(18)</td>
<td>Set or index name.</td>
</tr>
<tr>
<td>LFS</td>
<td>CHAR(1)</td>
<td>Leaf scan indicator, when ACMODE is I. This indicates whether data is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>retrieved by sequential access to index leaf pages.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'N'—No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Y'—Yes</td>
</tr>
<tr>
<td>SORTC</td>
<td>CHAR(1)</td>
<td>Composite sort type. A nonblank value in this field indicates an actual</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sort is required (data cannot be accessed in sort order).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'D'—Distinct</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'G'—Group</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'M'—Merge join</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'O'—Order by</td>
</tr>
<tr>
<td>SORTN</td>
<td>CHAR(1)</td>
<td>Inner sort type. This is an actual sort performed for the inner loop of a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>merge join</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'M'—Merge join</td>
</tr>
<tr>
<td>SUBQC</td>
<td>CHAR(1)</td>
<td>Subquery correlation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'N'—Not correlated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Y'—Correlated</td>
</tr>
</tbody>
</table>
Step types

Values in the STYPE column describe the type of processing:

<table>
<thead>
<tr>
<th>Step type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (Table access)</td>
<td>Access to a single table</td>
</tr>
<tr>
<td>2 (Nested loop join)</td>
<td>Join using linked constraint</td>
</tr>
<tr>
<td>3 (Merge join)</td>
<td>Join by scanning both tables and sorting the entire result</td>
</tr>
<tr>
<td>4 (Sort)</td>
<td>Sort required by an ORDER BY parameter</td>
</tr>
<tr>
<td>5 (Merge group)</td>
<td>Sorting required by an aggregate function on distinct column values with the grouped results</td>
</tr>
<tr>
<td>6 (OR list)</td>
<td>Sorting required by one or more OR operators in a WHERE clause</td>
</tr>
</tbody>
</table>

Alternatives to the Default ACCESS_PLAN Table

You can use SQL procedures to tailor the way you retrieve and present access strategy information. You can also:

- Create a table with the same column definitions (but, optionally, different column names) as in table ACCESS_PLAN and specify this table when you issue the EXPLAIN statement.
- Use the ALTER TABLE statement to add columns to table ACCESS_PLAN, or include additional columns at the end of the column list when you create a table equivalent to ACCESS_PLAN.
- Create one or more views of table ACCESS_PLAN.

Managing the Contents of an ACCESS_PLAN Table

Each time an EXPLAIN statement is executed; it inserts rows into an ACCESS_PLAN table. Periodically, contents of the table should be deleted using the DELETE statement.

Enhancing the Presentation of Access Strategy Information

Enhancing the Presentation of Access Strategy Information contains an SQL script with the definitions and data for a view that returns the access strategy information in an easy-to-read and understandable format.
Examples

Explaining RCMs in an Access Module

The following EXPLAIN statement returns the access strategy for each explainable statement in the EMPDSP01, EMPDSP02, and EMPDSP03 RCMs in the EMPAM001 access module. CA IDMS stores the access strategy in a table named EMPAM001_ACCESS.

```
explain access module empam001
    module empdsp01, empdsp02, empdsp03
    into table empam001_access;
```

Explaining a Specified Statement

The following EXPLAIN statement returns an access strategy for the specified SELECT statement. The access plan is identified by the reference number 4. By default, CA IDMS stores the access strategy in the ACCESS_PLAN table.

```
explain statement 'select e1.emp_id
    from employee e1, position p1
    where e1.emp_id = p1.emp_id
    and p1.salary_amount >
    (select p2.salary_amount
     from employee e2, position p2
     where e1.emp_id = e2.emp_id
     and e2.manager_id = p2.emp_id)'
    statement number 4;
```

FETCH

The FETCH data manipulation statement retrieves values from the result table associated with a cursor. You can use this statement only in SQL that is embedded in a program.

Authorization

None required.

Syntax

```
FETCH cursor-name
    INTO host-variable [ local-variable routine-parameter ]
    [ :dyn-buffer USING DESCRIPTOR descriptor-area-name ]
    [ :bulk-buffer bulk-options ]
    [ :dyn-buffer dynamic.bulk-options2 ]
```
**Expansion of bulk-options**

- START : `start-variable-name`
- ROWS : `row-count-variable-name`

**Expansion of dynamic-bulk-options2**

- START : `start-variable-name`
- ROWS : `row-count-variable-name`
- using sql DESCRIPTOR `descriptor-area-name`
Parameters

**cursor-name**

Specifies the cursor to be used for retrieving values. **Cursor-name** must identify an open cursor previously defined by a DECLARE CURSOR statement within the application program or by an ALLOCATE CURSOR statement executed within the same SQL transaction.

**INTO**

Directs CA IDMS to retrieve a single row from the result table associated with the named cursor and to return the column values into the specified locations.

**Note:** An INTO clause is required for SQL that is imbedded in host programs.

**host-variable**

Identifies the host variables to which CA IDMS is to assign values retrieved from a result table defined by a query expression. CA IDMS assigns the value in the first result column to the first host variable, the value in the second result column to the second host variable, and so on.

**Host-variable** must be a host variable declared previously in the host-language application program.

**Note:** In COBOL, **host-variable** can be a non-bulk structure. For more information, see the *CA IDMS SQL Programming Guide*.

You must specify the same number of host variables in the INTO parameter as the number of columns in the result table. Multiple host variables must be separated by commas. For expanded **host-variable** syntax, see Host Variables.

**local-variable**

**routine-parameter**

Identifies the local variable or routine parameter which CA IDMS is to assign values retrieved from a result table defined by a query expression. CA IDMS assigns the value in the first result column to the first local variable or routine parameter, the value in the second result column to the second local variable or routine parameter, and so on. You must specify the same number of local variables and routine parameters in the INTO parameter as the number of columns in the result table.
**:dyn-buffer**

Identifies a variable or a bulk buffer into which CA IDMS is to return all values retrieved from one or more rows of the result table associated with the named cursor.

*Dyn-buffer* must identify a variable previously declared in the host language application program or SQL routine.

The size of *dyn-buffer* must be sufficient to hold one row of the result table if specified as part of the INTO parameter or *row-count-variable* rows if specified as part of the BULK parameter. The format of the data returned into *dyn-buffer* is determined by the column descriptions in the SQL descriptor area specified in the USING DESCRIPTOR parameter.

**USING DESCRIPTOR**

Specifies the SQL descriptor area that describes the format in which the columns of the result table are to be returned to the host-language application program or SQL routine.

The specification of a descriptor area is a CA IDMS extension of the SQL standard.

*descriptor-area-name*

Directs CA IDMS to use the named area as the descriptor area. *Descriptor-area-name* must identify an SQL descriptor area.

For the layout of an SQL descriptor area, see SQL Descriptor Area.

**BULK**

Directs CA IDMS to retrieve one or more rows from the result table associated with the cursor and to return the column values into a contiguous storage area. The specification of BULK is a CA IDMS extension of the SQL standard.

**:bulk-buffer**

Identifies a variable to which CA IDMS is to assign values retrieved from one or more rows of the result table associated with the named cursor. *Bulk-buffer* must identify a variable previously declared in the host-language application program or SQL routine.

*Bulk-buffer* must be defined as a multiply-occurring structure having the same number of sub-elements in one occurrence as the number of columns in the result table.

**bulk-options**

Optionally specify the location in *bulk-buffer* for the first row fetched and/or the number of rows to be fetched from the result table associated with the cursor. Expanded syntax for *bulk-options* immediately follows the statement syntax.
**dynamic-bulk-options2**

Provides specifications for dynamically retrieving one or more rows from the result table associated with the named cursor.

Expanded syntax for **dynamic-bulk-options2** appears immediately following the expanded syntax for **bulk-options**. Descriptions of **dynamic-bulk-options2** parameters appear above.

**Note:** Dyn-buff, bulk-buffer, start-variable-name, and row-count-variable-name are variables that can be host variables or when the statement is used in an SQL routine, local variables or routine parameters. In this case, their names must not be preceded with a colon.

**Parameters for Expansion of bulk-options**

**START : start-variable-name**

Identifies a variable containing the relative position within the bulk buffer to which CA IDMS is to assign the values in the first row retrieved from the result table.

Values in subsequent rows of the result table are assigned sequentially to subsequent positions in the bulk buffer.

**Start-variable-name** must be a variable previously declared in the host-language application program. The value in the variable must be an integer in the range 1 through the number of rows that fit in the bulk buffer.

For languages whose subscript values are relative to 0, the value for **start-variable-name** must be in the range 0 through one less than the number of entries which fit in the bulk buffer.

If you do not specify the START parameter, CA IDMS assigns the values in the first row of the result table to the first row of the array.

**ROWS : row-count-variable-name**

Identifies a variable that specifies the maximum number of rows in the result table CA IDMS is to assign to the bulk buffer.

**Row-count-variable-name** must be a variable previously declared in the host-language application program. The value in the variable must be an integer in the range 1 through the number of rows that fit in the bulk buffer.

The ROWS parameter must be specified if a USING DESCRIPTOR clause is specified in a BULK parameter.

If you do not specify the ROWS parameter, CA IDMS assigns the rows in the result table to the buffer sequentially until no more rows exist in the result table or the buffer has been filled.
Parameters for Expansion of dynamic-bulk-options2

The following additional parameter is used with bulk-options to create dynamic-bulk-options2:

descriptor-area-name

Directs CA IDMS to use the named area as the descriptor area.
Descriptor-area-name must identify an SQL descriptor area.
Usage

Compatible Data Types

The data types of the values retrieved by the FETCH statement and the data types of the variables named in the INTO parameter must be compatible for assignment. If the values are assigned to a buffer defined as an array, the data types of the array elements must be compatible with the data types of the values.

FETCH Execution

When executing a FETCH statement, CA IDMS:
1. Positions a cursor on the next row following the current row
2. Retrieves one or more rows of values from the result table beginning with the new current row
3. Assigns the retrieved values to the specified variables
4. Leaves the cursor positioned on the last row retrieved

No More Rows

CA IDMS returns an SQLCODE value of 100 when any of the following is true:
- The result table associated with the cursor named in the FETCH statement is empty
- The result table associated with the cursor named in the FETCH statement is not empty, and the cursor is positioned after the last row before the statement is executed
- A bulk fetch operation retrieves fewer rows than are requested in the FETCH statement

In each case, CA IDMS leaves the cursor positioned after the last row.

Use of the Descriptor Area

When you use dynamic SQL to return data to a host-language application program in a form different from that in which it is stored in the database, you can modify the data characteristics in the SQL descriptor area named in the FETCH statement. You must make any changes to the descriptor area before the first fetch operation. You should not change the contents of the descriptor area after the first fetch operation and before the closing of the cursor.

Static and Dynamic Cursors

The format of the output of a static cursor is known at compile time. The format of the output of a dynamic cursor is often not known at compile time. Typically, you specify dyn-buffer when the cursor is dynamic, such as when the SELECT statement associated with the cursor is not known at compile time.
Examples

Fetching Multiple Rows

The following FETCH statement retrieves values from a result table defined by PROJ_CURSOR. Descriptions of the data in the output buffer are in a descriptor area named BUFF-1-SQLDA. The retrieved values are assigned to the CURSOR-BUFF-1 buffer, starting at the position in the buffer indicated by the value in BUFF-1-START. The value in BUFF-1-ROWS determines the number of rows retrieved.

EXEC SQL
  FETCH PROJ_CURSOR
    BULK :CURSOR-BUFF-1
    START :BUFF-1-START
    ROWS :BUFF-1-ROWS
    USING DESCRIPTOR BUFF-1-SQLDA
END-EXEC

Fetching a Single Row

The following FETCH statement retrieves values from one row of the BONUS_CURSOR cursor. The values are assigned to the host variables EMP-ID and BONUS-AMT which have an associated indicator variable.

EXEC SQL
  FETCH BONUS_CURSOR
    INTO :EMP-ID, :BONUS-AMT :BONUS-IND
END-EXEC

More Information

- For more information about defining and manipulating cursors, see CLOSE, DECLARE CURSOR, and OPEN.
- For more information about host variables, local variables or routine parameters, see Host Variables, Local Variables (see page 81) or Routine Parameters.
- For more information about compatible data types for assignment operations, see Comparison, Assignment, Arithmetic, and Concatenation Operations.
- For more information about the SQL descriptor area, see the CA IDMS Navigational DML Programming Guide.

GET DIAGNOSTICS

The GET DIAGNOSTICS statement extracts information on exception or completion conditions of the last executed SQL statement from the diagnostics area and returns it to the issuer. Use this statement in SQL that is embedded in a program.
Syntax

```
GET DIAGNOSTICS ▶- statement-info ▼- condition-nr condition-info ◄
```

**Expansion of statement-info**

```
```

**Expansion of condition-info**

```
```
Parameters

A routine-parameter, host-variable, or local-variable must be specified for each statement-info or condition-info item.

**statement-info**

Identifies the type of statement information to be extracted and returned. Statement-info names that begin with 'IDMS_' are extensions to the SQL standard.

**CONDITION**

Requests diagnostic information for a condition.

**condition-nr**

 Specifies the number of the completion or exception condition for which diagnostics information is being requested. An exception is raised if condition-nr does not refer to a valid condition number.

**condition-info**

Identifies the type of condition-related information to be extracted and returned. Condition-info names that begin with 'IDMS_' are extensions to the SQL standard.

**EXCEPTION**

Specifies a synonym for CONDITION. While it is part of the current SQL standard, its use is discouraged because it will not be in future SQL standards.

**Parameters for Expansion of statement-info**

**routine-parameter**

Identifies an SQL routine parameter that is to receive the value of the specified diagnostics item. Routine-parameter must be a parameter of the current SQL routine and must be compatible for assignment with the specified diagnostic item.

See Expansion of Routine-parameter for information about expanded syntax.

**host-variable**

Identifies a host variable that is to receive the value of the specified diagnostics item. Host-variable must be a host variable previously declared in the application program and must be compatible for assignment with the specified diagnostic item.

See Expansion of Host-variable for information about expanded syntax.
**local-variable**

Identifies a local variable of an SQL routine that is to receive the value of the specified diagnostics item. **Local-variable** must be a local variable declared in the current SQL routine and must be compatible for assignment with the specified diagnostic item.

See Expansion of Local-variable for information about expanded syntax.

**COMMAND_FUNCTION**

Returns a value with data type varchar (64) indicating the type of SQL command that was last executed. The values that may be returned are listed under the Statement Type column in Table Procedure Requests.

**COMMAND_FUNCTION_CODE**

Returns a value with data type integer indicating the type of SQL command that was last executed. The values that may be returned are listed under the Command Number column in Table Procedure Requests.

**DYNAMIC_FUNCTION**

Returns a value with data type varchar (64) indicating the type of SQL command that was prepared or dynamically executed by the last command. The values that may be returned are listed under the Statement Type column in Table Procedure Requests.

**DYNAMIC_FUNCTION_CODE**

Returns a value with data type integer indicating the type of SQL command that was prepared or dynamically executed by the last command. The values that may be returned are listed under the Command Number column in Table Procedure Requests.

**IDMS_RETURNED_RESULT_SETS**

Returns a value with data type integer indicating the number of result sets returned by a procedure invoked by the last command. This value is only valid if the diagnosed statement is a call or select of an SQL invoked procedure.

**MORE**

Returns a value with data type char(1). A value of 'Y' indicates that the execution of the previous SQL statement caused more conditions than have been set in the diagnostics area. A value of 'N' means that the diagnostics area contains information on all the completion and exception conditions.

**NUMBER**

Returns a value with data type integer indicating the number of the exceptions or completion conditions set by the execution of the previous SQL statement for which information is available in the diagnostics area.
ROW_COUNT

Returns a value with data type DEC(31). The value depends on the type of the previously executed statement:

- INSERT - Number of rows inserted
- DELETE - Number of rows deleted
- UPDATE - Number of rows updated
- BULK FETCH - Number of rows fetched
- FETCH - 1 or 0

Parameters for Expansion of condition-info

IDMS_MESSAGE_COMMENTS

Returns a value with data type varchar(4000) containing the comments in the message dictionary for the message associated with the condition.

IDMS_MESSAGE_DEFINITION

Returns a value with data type varchar(4000) containing the definition in the message dictionary of the message associated with the condition.

IDMS_MESSAGE_ID

Returns a value with data type char(8) containing the message ID in the message dictionary of the message associated with the condition.

IDMS_MODULE_NUMBER

Returns a value with data type integer containing the number of the module that detected the condition.

IDMS_REASON_CODE

Returns a value with data type integer containing the reason code of the condition.

IDMS_SQLCODE

Returns a value with data type integer containing the SQLCODE value associated with the condition.

IDMS_TASK_ID

Returns a value with data type integer containing the IDMS task ID of the task that encountered the condition.

MESSAGE_LENGTH

Returns a value with data type integer indicating the length of the message associated with the specified condition.
MESSAGE_TEXT

Returns a value with data type varchar(256) containing the message text associated with the specified condition.

RETURNED_SQLSTATE

Returns a value with data type char(5) indicating the SQLSTATE associated with the specified condition.
Example

The procedure TGETDIAG1 executes a SELECT statement that causes a number of string truncations. The first GET DIAGNOSTICS returns the number of conditions that the SELECT statement raised. A WHILE LOOP containing the second GET DIAGNOSTICS concatenates the message texts of all the raised conditions to the RESULT parameter of the procedure.

```sql
set options command delimiter '++';
create procedure SQLROUT.TGETDIAG1
  ( TITLE    varchar(10) with default
  , P_NAME   char(18)
  , P_NUMBER integer
  , RESULT   varchar(512)
 )
  EXTERNAL NAME TGETDIAG LANGUAGE SQL
begin not atomic
  declare L_NUMBER  integer      default 1;
  declare L_MESSAGE varchar(256) default ' ';
  select NAME into P_NAME from system.schema
    where cast(NAME as char(12)) = P_NAME;
  /* retrieve the number of conditions raised */
  get diagnostics P_NUMBER = NUMBER;
  while (L_NUMBER <= P_NUMBER)
  do
    /* retrieve the message text of the raised condition */
    get diagnostics condition L_NUMBER
      L_MESSAGE = MESSAGE_TEXT
    set RESULT = RESULT || ' ' || L_MESSAGE;
    set L_NUMBER = L_NUMBER + 1;
  end while;
++
  commit++
set options command delimiter default++
call SQLROUT.TGETDIAG1('TGETDIAG1', 'SYSTEM');
++
  TITLE    P_NAME                 P_NUMBER
  -----    ------                 -------
  TGETDIAG1   SYSTEM                        4
++
  RESULT
  ------
  DB001043 T171 C1M322: String truncation DB001043 T171 C1M322:
  String truncation DB001043 T171 C1M322: String truncation
  DB001043 T171 C1M322: String truncation
```
GET STATISTICS

The GET STATISTICS statement returns statistical information for the current transaction. It is a CA IDMS extension to the SQL standard. Use this statement in SQL that is embedded in a program, in the SQL command facility, and in the command console of CA IDMS Visual DBA.

Syntax

```
GET STATISTICS transaction-info
```

Expansion of transaction-info

```
routine-parameter
host-variable
local-variable
```

- SQL_COMMANDS
- ROWS_Fetched
- ROWS_INSERTED
- ROWS_UPDATED
- ROWS_DELETED
- SORT
- ROWS_SORTED
- MIN_ROWS_SORTED
- MAX_ROWS_SORTED
- AM_RECOMPILES
- PAGES_READ
- PAGES_WRITTEN
- PAGES_REQUESTED
- CALC_TARGET
- CALC_OVERFLOW
- VIA_TARGET
- VIA_OVERFLOW
- RECORDS_REQUESTED
- RECORDS_RELOADED
- CALLS_DBMS
- FRAGMENTS_STORED
- RECORDS_RELOCATED
- TOTAL_LOCKS
- SHARE_LOCKS_HELD
- NON_SHARE_LOCKS_HELD
- TOTAL_LOCKS_FREED
- SR8_SPLITS
- SR8_SPAWNS
- SR8_STORED
- SR8_ERASED
- SR7_STORED
- SR7_ERASED
- B_TREE_SEARCH
- B_TREE_LEVELS_SEARCH
- ORPHANS_ADOPTED
- LEVELS_SEARCH_BEST_CASE
- LEVELS_SEARCH_WORST_CASE
- RECORDS_UPDATED
- SHARE_LOCKS_ACQ_CALL
- SHARE_LOCKS_FREED_CALL
- NON_SHARE_LOCKS_ACQ_CALL
- NON_SHARE_LOCKS_FREED_CALL
**Parameters**

**routine-parameter**

Identifies an SQL routine parameter that is to receive the value of the specified statistics item. **Routine-parameter** must be a parameter of the current SQL routine and must be compatible for assignment with the specified statistics item.

See Expansion of Routine-parameter for information about expanded syntax.

**host-variable**

Identifies a host variable that is to receive the value of the specified statistics item. **Host-variable** must be a host variable previously declared in the application program and must be compatible for assignment with the specified statistics item.

For more information about expanded syntax, see Expansion of Host-variable.

**local-variable**

Identifies a local variable of an SQL routine that is to receive the value of the specified statistics item. **Local-variable** must be a local variable declared in the SQL-invoked routine and must be compatible for assignment with the specified statistics item.

For more information about expanded syntax, see Expansion of Local-variable.

**Note:** A routine-parameter, host-variable or local-variable must be specified for each transaction-info when the statement is embedded in a program. Otherwise, these must not be specified.

**transaction-info**

Identifies the type of transaction information that is to be returned. Each item has an integer data type and represents statistical information for the current transaction.

**Note:** For more information about these items, see the DCMT DISPLAY STATISTICS SYSTEM and DCMT DISPLAY TRANSACTION commands in the *CA IDMS System Tasks and Operator Commands Guide*.

* Requests that all **transaction-info** items are to be retrieved. This is not allowed in combination with the specification of a **routine-parameter**, **host-variable**, or **local-variable** and therefore cannot be used in a program.
Example

The SQL procedure TGETSTA1 counts the number of rows of one of four tables:

- SYSTEM.TABLE
- SYSTEM.COLUMN
- SYSTEM.SCHEMA
- DEMOEMPL.EMPLOYEE

The actual table is selected through the value of the TITLE parameter. Besides returning the count of rows, the procedure also returns the values of a number of statistical information items for the transaction:

- SQL_COMMANDS
- PAGES_REQUESTED
- PAGES_READ
- CALLS_DBMS
- TOTAL_LOCKS

```
set options command delimiter '++';
drop procedure SQLROUT.TGETSTA1++
create procedure SQLROUT.TGETSTA1
( TITLE             char(8) with default
 , P_COUNT           integer
 , P_SQL_COMMANDS    integer
 , P_PAGES_REQUESTED integer
 , P_PAGES_READ      integer
 , P_CALLS_DBMS      integer
 , P_TOTAL_LOCKS     integer
 )
EXTERNAL NAME TGETSTA1 LANGUAGE SQL
Lab1: begin not atomic
case TITLE
  when 'TABLE'
    then select count(*) into P_COUNT
        from SYSTEM.TABLE;
  when 'COLUMN'
    then select count(*) into P_COUNT
        from SYSTEM.COLUMN;
  when 'SCHEMA'
    then select count(*) into P_COUNT
        from SYSTEM.SCHEMA;
  when 'EMPLOYEE'
    then select count(*) into P_COUNT
        from DEMOEMPL.EMPLOYEE;
end case;
```
get statistics
    P_SQL_COMMANDS = sql_commands
    , P_PAGES_REQUESTED = pages_requested
    , P_PAGES_READ = pages_read
    , P_CALLS_DBMS = calls_dbms
    , P_TOTAL_LOCKS = total_locks;
end
++
set options command delimiter default ++

call sqlrout.TGETSTA1('TABLE');
++
*+ TITLE         P_COUNT  P_SQL_COMMANDS  P_PAGES_REQUESTED  P_PAGES_READ
*+ 808               2                836             9
*+ P_CALLS_DBMS  P_TOTAL_LOCKS
*+ 813           1673

call sqlrout.TGETSTA1('COLUMN');
++
*+ TITLE         P_COUNT  P_SQL_COMMANDS  P_PAGES_REQUESTED  P_PAGES_READ
*+ 6450               3               8953           1068
*+ P_CALLS_DBMS  P_TOTAL_LOCKS
*+ 8071           8300

call sqlrout.TGETSTA1('SCHEMA');
++
*+ TITLE         P_COUNT  P_SQL_COMMANDS  P_PAGES_REQUESTED  P_PAGES_READ
*+ 56               4                 59              2
*+ P_CALLS_DBMS  P_TOTAL_LOCKS
*+ 61            130

call sqlrout.TGETSTA1('EMPLOYEE');
++
*+ TITLE         P_COUNT  P_SQL_COMMANDS  P_PAGES_REQUESTED  P_PAGES_READ
*+ 55               5                 58              2
*+ P_CALLS_DBMS  P_TOTAL_LOCKS
*+ 60            128
GRANT Access Module Execution Privilege

The GRANT Access Module Execution Privilege authorization statement gives one or more users or groups the privilege of executing a specified access module. The GRANT EXECUTE statement is a CA IDMS extension of the SQL standard.

Authorization

To grant access module execution privilege, you must own the access module, hold grantable privilege on the access module, or hold DBADMIN privilege on the dictionary that contains the access module.

Syntax

```
GRANT EXECUTE
ON ACCESS MODULE access-module-name
TO PUBLIC
WITH GRANT OPTION
```
Parameters

ON ACCESS MODULE access-module-name
    Specifies the access module to which the EXECUTE privilege applies.
    
    schema-name
    Identifies the schema associated with access-module-name.
    If you do not specify schema-name, it defaults to the current schema in effect for your SQL session.

TO
    Identifies the users to whom you are giving the EXECUTE privilege.

PUBLIC
    Specifies all users.

authorization-identifier
    Identifies a user or group. For expanded authorization-identifier syntax, see Expansion of Authorization-identifier.

WITH GRANT OPTION
    Gives the privilege of granting the EXECUTE privilege on the named access module to the users identified in the TO parameter. The owner of the resource, a holder of the applicable DBADMIN privilege, or a holder of SYSADMIN privilege can specify WITH GRANT OPTION
    A privilege granted with the WITH GRANT OPTION is called a grantable privilege.
Usage

Multiple Access Modules in One GRANT EXECUTE Statement

You can grant privileges on multiple access modules in a single GRANT statement by using an asterisk (*) as a wildcard character. A wildcard character represents one or more characters omitted from a string.

If used, the asterisk must be the last character in access-module-name. If access-module-name ends with an asterisk, the name represents all the access modules whose names match the pattern established.

For example, the access module name TST* in a GRANT EXECUTE statement represents all access modules whose names start with TST in the specified or current schema.

Duration of Privileges

Users hold the EXECUTE privilege granted on an access module until the privilege is explicitly taken away by means of the REVOKE EXECUTE statement.

Privileges Granted to Groups

When you grant a privilege to a group, each user in the group holds the privilege. If you subsequently add a user to the group, that user also holds the privilege. If you drop a user from the group, that user longer holds the privilege.

Access Modules in Categories

Although you can grant the EXECUTE privilege on a specified access module, the more typical method is to grant the privilege on a category that includes multiple access modules. The use of categories for granting the EXECUTE privilege reduces overhead and provides better performance in a multiuser environment.

Executing an Access Module

To execute an access module, the owner of the access module must have the authority to execute all statements in the access module.

To execute an access module, a user other than the owner must:

- Hold the EXECUTE privilege on the access module
- If CA IDMS internal security is in effect, the owner of the access module must have the right to grant the privileges necessary to execute the SQL statements in the access module
- If external security is in effect, the executing user must have the authority to execute statements in the access module
Example

Granting the EXECUTE Privilege

The following GRANT EXECUTE statement gives the EXECUTE privilege on all access modules in the HR schema that begin with EMP to the users in the groups PER_GRP_1 and PER_GRP_2:

grant execute
on access module hr.emp*
  to per_grp_1, per_grp_2;

More Information

- For more information about revoking the EXECUTE privilege, see REVOKE Execution Privilege.
- For more information about CA IDMS internal security and about external security for CA IDMS resources, see the CA IDMS Security Administration Guide.

GRANT Definition Privileges

The GRANT Definition Privileges authorization statement gives one or more users the privilege of performing selected actions on a specified access module, schema, table, view, procedure or table procedure. It is also a CA IDMS extension of the SQL standard.

Authorization

To issue a GRANT statement for a definition privilege, you must own the resource, hold grantable privilege on the resource, or hold DBADMIN privilege on the dictionary containing the definition.

Syntax
**Parameters**

**DEFINE**
Gives the ALTER, CREATE, DISPLAY, DROP, and REFERENCES privileges, as applicable on the resource identified in the ON parameter to the users or groups identified in the TO parameter.

**ALTER**
Gives the ALTER privilege on resource identified in the ON parameter to the users or groups identified in the TO parameter.

The ALTER privilege on a resource allows you to modify the definition of the resource.

**CREATE**
Gives the CREATE privilege on the resource identified in the ON parameter to the users or groups identified in the TO parameter.

The CREATE privilege on a resource allows you to define the resource.

**DISPLAY**
Gives the DISPLAY privilege on the resource identified in the ON parameter to the users or groups identified in the TO parameter.

The DISPLAY privilege on an access module allows you to execute the EXPLAIN statement on the access module.

**DROP**
Gives the DROP privilege on the resource identified in the ON parameter to the users or groups identified in the TO parameter.

The DROP privilege on a resource allows you to delete the definition of the resource.

**REFERENCES**
Gives the REFERENCES privilege on the resource identified in the ON parameter to the users or groups identified in the TO parameter.

The REFERENCES privilege on a table allows a user to define referential constraints in which the table is the referenced table.

**ON**
Specifies the resource to which the definition privileges apply.
**ACCESS MODULE access-module-name**

Specifies that the privileges apply to any version of `access-module-name` in the associated schema.

**schema-name**

Identifies the schema associated with `access-module-name`.

If you do not specify `schema-name`, it defaults to the current schema in effect for your SQL session.

**SCHEMA schema-name**

Identifies an SQL schema.

**table table-name**

Identifies a table, view, procedure or table procedure.

If `table-name` does not include schema name qualifier, the schema name qualifier defaults to the current schema in effect for your SQL session.

**schema-name**

Specifies the schema with which the function identified by function-identifier is associated. If schema-name is not specified, the schema defaults to the current schema in effect for your SQL session.

**function-identifier**

Identifies the function.

**TO**

Identifies the users to whom you are giving the definition privileges.

**PUBLIC**

Specifies all users.

**authorization-identifier**

Identifies a user or group. For expanded `authorization-identifier` syntax, see Expansion of Authorization-identifier.

**WITH GRANT OPTION**

Gives the privilege of granting the specified definition privileges on the named resource to the users identified in the TO parameter. Only the owner of the resource or a user holding the DBADMIN privilege can specify WITH GRANT OPTION.

A privilege granted with the WITH GRANT OPTION is called a grantable privilege.
Usage

Multiple Entities in One GRANT Statement

You can grant privileges on multiple entities of the same type in a single GRANT statement by using an asterisk (*) as a wildcard character. A wildcard character represents one or more characters omitted from a string.

If used, the asterisk must be the last character in the resource name. The asterisk can replace all or part of:

- **Access-module-name**
- **Schema-name** on the SCHEMA parameter
- **Table-identifier, view-identifier, procedure-identifier or table-procedure-identifier in table-name**
- **Function-identifier**

A resource name with an asterisk represents all the entities of the same type whose names match the pattern established by the name with the asterisk. For example, the access module name ACC* in a GRANT statement represents all access modules whose names start with ACC in the specified or current schema.

The DEFINE Keyword

When you use the DEFINE keyword with a GRANT statement, you grant a set of definition privileges on a resource to one or more users or groups.

When you use the DEFINE keyword with a REVOKE statement, you revoke any definition privileges that have been previously granted on the resource from the specified users or groups.

This means that if you GRANT CREATE privilege on a table, you can revoke the privilege with a REVOKE SELECT statement or a REVOKE DEFINE statement. Using REVOKE DEFINE is an efficient technique when you intend to revoke all definition privileges on a table from a user or group, whether the privileges were granted singly or as a set.

Similarly, you can GRANT DEFINE on a table to a user and then REVOKE DELETE on the table from the same user as a way to grant all but one definition privilege.
Duration of Privileges

Users hold privileges granted on a resource until the privileges are explicitly taken away by means of the REVOKE statement.

Privileges Granted to Groups

When you grant a privilege to a group, each user in the group holds the privilege. If you subsequently add a user to the group, that user also holds the privilege. If you drop a user from the group, that user longer hold the privilege.

Example

Granting Privileges on a Schema

The following GRANT statement gives the ALTER, CREATE, DISPLAY, and DROP privileges on all schemas that begin with DSF to user DSF. The statement also gives user DSF the privilege of granting the same privileges to other users.

```
grant define
    on schema dsf*
    to dsf
    with grant option;
```

More Information

- For more information about definition privileges, see REVOKE SQL Definition Privileges.
- For more information about granting privileges, see your security administrator.

GRANT Table Access Privileges

The GRANT Table Access Privileges authorization statement gives one or more users or groups the privilege of performing selected actions on a specified table, view, function, procedure or table procedure.

Authorization

To issue a GRANT statement for a table privilege, you must own the table, view, function, procedure, or table procedure, hold the corresponding grantable privilege on the table, view, procedure or table procedure, or hold the DBADMIN privilege on the database that contains the table, view, function, procedure, or table procedure.
Syntax

```
GRANT ACCESS
  ┌──┬───┐
  │   │   │
  │ DELETE │ INSERT │ SELECT │ UPDATE │
  └───┬───┘
      │
      └───┬───┐
        │   │
        │ ON table │ table-name │
        │         │ schema-name │
        │         │ function-identifier │
        │         │
        └───┬───┘
          │
          └───┬───┐
            │   │
            │ TO PUBLIC │
            │ authorization-identifier │
            │
            └───┬───┘
              │
              └───┬───┘
                │
                └───┬───┐
                  │
                  └───┬───┘
                    │
                    └───┬───┘
                      │
                      └───┬───┘
                        │
                        └───┬───┘
                          │
                          └───┬───┘
                            │
                            └───┬───┘
                              │
                              └───┬───┘
                                │
                                └───┐
                                  │
                                  └───┘
```

with GRANT OPTION
Parameters

ACCESS
Gives the DELETE, INSERT, SELECT, and UPDATE privileges on the table, view, function, procedure or table procedure identified in the ON parameter to the users or groups identified in the TO parameter.

The ACCESS parameter is a CA IDMS extension of the SQL standard.

DELETE
Gives the DELETE privilege on the table, view, or table procedure identified in the ON parameter to the users or groups identified in the TO parameter.

The DELETE privilege on a table, view, or table procedure allows you to delete rows from the table or view.

INSERT
Gives the INSERT privilege on the table, view, or table procedure identified in the ON parameter to the users or groups identified in the TO parameter.

The INSERT privilege on a table, view, or table procedure allows you to insert rows into the table or view.

SELECT
Gives the SELECT privilege on the table, view, function, procedure or table procedure identified in the ON parameter to the users or groups identified in the TO parameter.

The SELECT privilege on a table, view, function, procedure or table procedure allows you to:
- Retrieve data from the table, view, function, procedure or table procedure
- Name the table, view, function, procedure or table procedure in a subquery

UPDATE
Gives the UPDATE privilege on the table, view, or table procedure identified in the ON parameter to the users or groups identified in the TO parameter.

The UPDATE privilege on a table, view, or table procedure allows you to modify data in the table or through the view.

ON table table-name
Identifies the table, view, procedure or table procedure to which the table access privileges apply.

If table-name does not include schema name qualifier, the schema name qualifier defaults to the current schema in effect for your SQL session.

The optional keyword TABLE is a CA IDMS extension of the SQL standard. See Expansion of Table-name for expanded table-name syntax.
schema-name
Optional qualifier of the function-identifier. If not specified the schema name qualifier defaults to the current schema in effect for your SQL session.

function-identifier
Identifies the function to which the access privilege applies.

TO
Identifies the users to whom you are giving table access privileges.

PUBLIC
Specifies all users.

authorization-identifier
Identifies a user or group. For expanded authorization-identifier syntax, see Expansion of Authorization-identifier.

WITH GRANT OPTION
Gives the privilege of granting the specified privileges on the named table, view, procedure or table procedure to the users identified in the TO parameter. The owner of the resource, a holder of the applicable DBADMIN privilege, or a holder of SYSADMIN privilege can specify WITH GRANT OPTION.

A privilege granted with the WITH GRANT OPTION is called a grantable privilege.
Usage

Multiple Tables and Views in One GRANT Statement

You can grant privileges on multiple tables, views, functions, procedures and table procedures in a single GRANT statement by using an asterisk (*) as a wildcard character. A wildcard character represents one or more characters omitted from a string.

If used, the asterisk must be the last character in the table, view, function, procedure or table procedure identifier in table-name or function-identifier. A table, view, function, procedure or table procedure identifier with an asterisk represents all the tables, views, and table procedures whose identifiers match the pattern established by the identifier with the asterisk.

For example, the table, view, function, procedure or table procedure identifier EST* in a GRANT statement represents all tables, views, procedures and table procedures whose identifiers start with EST in the specified or current schema.

Duration of Privileges

Users hold privileges granted on a table, view, function, procedure or table procedure until the privileges are explicitly taken away by means of the REVOKE statement.

The ACCESS Keyword

When you use the ACCESS keyword with a GRANT statement, you grant a set of access privileges on a table, view, function, procedure or table procedure to one or more users or groups.

When you use the ACCESS keyword with a REVOKE statement, you revoke any access privileges that have been previously granted on the table, view, function, procedure or table procedure from the specified users or groups.

This means that if you GRANT SELECT privilege on a table, you can revoke the privilege with a REVOKE SELECT statement or a REVOKE ACCESS statement. Using REVOKE ACCESS is an efficient technique when you intend to revoke all access privileges on a table from a user or group, whether the privileges were granted singly or as a set.

Similarly, you can GRANT ACCESS on a table to a user and then REVOKE DELETE on the table from the same user as a way to grant all but one table access privilege.

Privileges Granted to Groups

When you grant a privilege to a group, each user in the group holds the privilege. If you subsequently add a user to the group, that user also holds the privilege. If you drop a user from the group, that user no longer holds the privilege.
Example

Granting Selected Privileges on a Table

The following GRANT statement gives the SELECT and UPDATE privileges on the EMPLOYEE table to users KRP, SAE, and PGD:

grant select, update
    on employee
    to krp, sae, pgd;

More Information

- For more information about revoking table access privileges, see REVOKE Table Access Privileges.
- For more information about granting privileges, see your security administrator.

INCLUDE

The INCLUDE precompiler directive statement directs the precompiler to create host variable definitions for a specified structure or table in the application program. You can use this statement only in SQL that is embedded in a program.

The INCLUDE statement is a CA IDMS extension of the SQL standard.

Authorization

To issue an INCLUDE statement that specifies:

- A table, you must own or have the SELECT privilege on the table
- SQLCA or SQLDA no privileges are required

Syntax

```
#include

SQLCA SQLDA TABLE

schema-name.

module-name

(column-count column-name)

AS descriptor-area-name
```

Expansion of sqlda-options

```

(column-count column-name)

AS descriptor-area-name
```
Expansion of table-options

\[
\begin{align*}
&\text{(column-name)} \quad \text{AS (column.alias)} \\
&\text{AS structure-name} \\
&\text{NO STRUCTURE} \\
&\text{NUMBER OF ROWS row-count} \\
&\text{PREFIX 'column-prefix'} \\
&\text{SUFFIX 'column-suffix'} \\
&\text{LEVEL 'level-number'}
\end{align*}
\]
Parameters

SQLCA
Directs the precompiler to define host variables for the fields in the SQL Communication Area (SQLCA) in the application program.

SQLDA
Directs the precompiler to define host variables for the fields in the SQL descriptor area in the application program.

sqlda-options
Specifies additional characteristics for the SQL descriptor area. Expanded syntax for sqlda-options is shown immediately following the INCLUDE syntax and documented directly below.

NUMBER OF COLUMNS column-count
Directs CA IDMS to build an SQL descriptor area large enough to contain the specified number of columns.

Column-count must be an integer in the range 1 through 1,024. The default is 100.

AS descriptor-area-name
Assigns the specified name to the SQL descriptor area being defined. Descriptor-area-name must be unique within the application program and must follow the conventions for host variable names.

If you do not specify a name for the SQL descriptor area, the name of the descriptor area is SQLDA.

TABLE table-name
Directs the precompiler to define host variables corresponding to one or more columns in the named table, view, procedure or table procedure in the application program.

Table-name must identify a base table, view, procedure or table procedure defined in the dictionary.

If table-name does not include an explicit schema name, the precompiler:

- Uses the schema name specified in the schema precompiler option, if any
- Returns an error if the schema precompiler option is not specified

For expanded table-name syntax, see Expansion of Table-name.
schema-name

Specifies the schema with which the function identified by function-identifier is associated. If schema-name is not specified, the precompiler:

- Uses the schema name specified in the schema precompiler option, if any
- Returns an error if the schema precompiler option is not specified

function-identifier

Directs the precompiler to define host variables corresponding to one or more columns in the named function in the application program.

table-options

Specifies additional information about the host variables to be defined for the named table, view, function, procedure, or table procedure. Expanded syntax for table-options is shown above immediately following the expanded syntax for sqlda-options and documented directly below.

module-name

Includes source statements from a module stored in the data dictionary into the source program.

The unmodified module is placed into the program by the precompiler at the location of the request. The module can, but need not, contain SQL or navigational DML statements. Any such statements are examined and expanded within the context of the program as if they were coded directly.

Note: INCLUDE module-name is equivalent to COPY IDMS module-name as documented in the CA IDMS DML Reference manuals.

The version of the module that is included is established as follows:

The version defaults to the highest version number defined in the data dictionary for the language mode under which the program is being compiled (for example, BATCH or IDMS-DC).

If no mode-specific version exists for module-name, the non-mode-specific version, if present, is included. If neither a mode-specific entry or a non-mode-specific entry for module-name has been established, an error results. The same rules apply to the module’s language where version-number defaults to the highest value defined in the data dictionary for the language in which the program is written.

Parameters for Expansion of sqlda-options

(column-name)

Specifies one or more columns for which the precompiler is to define host variables. Column-name must identify a column in the named table, view, function, procedure, or table procedure and must be unique within the list of column names.

If you specify one or more columns, the precompiler defines host variables only for the columns you name. If you do not specify any columns, the precompiler defines host variables for all the columns in the named table, view, or table procedure.
AS (column-alias)

Specifies names to be used for the host variables defined for the named columns. Column-alias must follow the conventions for host variable names.

You must specify no column aliases or the same number of column aliases as the number of host variables defined for columns in the named table, view, or table procedure.

If you do not specify any column aliases, the name used for each host variable is the name of the column for which the host variable is defined.

Parameters for Expansion of table-options

AS structure-name

Assigns the specified name to the data structure made up of the host variables corresponding to the columns in the named table, view, or table procedure. Structure-name must be a 1- through 31-character name that follows the conventions for host variable names.

If you do not specify a name for the data structure, and you do not specify NO STRUCTURE, the name of the structure is the name of the table or view used to define the structure.

NO STRUCTURE

Directs the precompiler not to group the host variable definitions together in a single data structure.

NUMBER OF ROWS row-count

Directs the precompiler to define an array, suitable for bulk processing, in which the host variables are sub-elements of a structure that occurs the specified number of times. Row-count should be an integer in the range from 2 to the largest number of entries supported for an array by the host language compiler. You must specify NUMBER OF ROWS to create a host variable array using INCLUDE.

If you specify NO STRUCTURE, NUMBER OF ROWS is ignored.

PREFIX 'column-prefix'

Specifies a character string to be used as the first portion of the name of each host variable defined for a column in the named table, view, procedure or table procedure. Column-prefix must be a one- through seven-character string and must be enclosed in single quotation marks.

The column prefix concatenated with the column name or alias and the column suffix, if specified, must follow the conventions for host variable names.
**SUFFIX 'column-suffix'**

Specifies a character string to be used as the last portion of the name of each host variable defined for a column in the named table, view, procedure or table procedure. *Column-suffix* must be a one- through seven-character string and must be enclosed in single quotation marks.

The column prefix, if specified, and the column name or alias concatenated with the column suffix must follow the conventions for host variable names.

**LEVEL level-number**

Directs the precompiler to assign the specified level number to the data structure as a whole or, when the INCLUDE statement specifies NO STRUCTURE, to each host variable defined for a column. *Level-number* must be an integer in the range 01 through 47.

If *level-number* is not specified, the default is 01.
Usage

SQLCA, SQLCODE, and SQLSTATE

An application program can contain an INCLUDE SQLCA statement or a host variable definition for SQLCODE and/or SQLSTATE within an SQL declaration section. In either case, the precompiler comments out the item and inserts the host variable definitions for the fields in the SQLCA.

If you include more than one of these items in an application program, the precompiler comments out all of them, inserts the host variable definitions for the fields in the SQLCA only once, and returns a warning.

If you do not include any of the items in a COBOL application program, the precompiler automatically creates the host variable definitions for the fields in the SQLCA if the application program contains SQL statements.

Note: This does not apply to PL/I application programs. A PL/I program must contain an INCLUDE SQLCA or a host variable definition for SQLCODE or SQLSTATE.

Multiple SQL Descriptor Areas

Each SQL descriptor area included in an application program must have a unique name. If you include multiple SQL descriptor areas, you can use the default name SQLDA for only one of the areas. You must use the AS parameter to specify a different name for each of the others.

Placement of INCLUDE TABLE

The INCLUDE TABLE statement may appear anywhere within the application program that variable declarations are allowed. However, to reference the generated variables as host variable in an SQL statement, you must place the INCLUDE TABLE statement within an SQL declaration section.

Column Names as Host Variable Names

You can use the names of the columns in an included table as host variable names only when the column names follow the conventions for host variable names. Otherwise, you must specify a column alias for each column for which you want to define a host variable.

Note: In a COBOL application program, the precompiler automatically converts underscores to hyphens in column names used as host variable names.

Indicator Variable Names

If a column within an included table is nullable, an indicator variable is generated for the column. The name of the indicator variable is the same as that of the data variable (column name or column alias) suffixed with ‘-I’.
Examples

Including the SQLCA

The following INCLUDE statement directs the precompiler to define host variables for the fields in the SQL Communication Area (SQLCA):

EXEC SQL
  INCLUDE SQLCA
END-EXEC

Including an SQL Descriptor Area

The following INCLUDE statement directs the precompiler to define host variables for the fields in an SQL descriptor area named BUFF-1-SQLDA. The maximum number of columns that can be described using BUFF-1-SQLDA is 30.

EXEC SQL
  INCLUDE SQLDA
  NUMBER OF COLUMNS 30
  AS BUFF-1-SQLDA
END-EXEC

Including a Table

The following INCLUDE statement directs the precompiler to define host variables corresponding to the named columns in the INSURANCE_PLAN table. The host variables are defined as a 10-row array named INS-COST-BUFFER.

EXEC SQL
  INCLUDE TABLE INSURANCE_PLAN
  AS INS-COST-BUFFER
    (PLAN_CODE, COMP_NAME, MAX_LIFE_COST, FAMILY_COST, DEP_COST)
    AS (PLANCODE, COMPNAME, MAXLIFE, FAMCOST, DEPCOST)
  NUMBER OF ROWS 10
END-EXEC

The previous statement generates COBOL variable declarations in this format:

01 INS-COST-BUFFER
  02 INS-COST-BUFFER-BULK OCCURS 10.
    03 PLANCODE ...
    03 COMPNAME ...
    03 MAXLIFE ...
    03 FAMCOST ...
    03 DEPCOST ...
More Information

- For more information about the SQLCA and the SQLDA, see SQL Communication Area and SQL Descriptor Area.
- For more information about host variables, see Host Variables.
- For more information about creating host variable array using INCLUDE, see the CA IDMS SQL Programming Guide.

**INSERT**

The INSERT data manipulation statement adds one or more rows to a table.

**Authorization**

To issue an INSERT statement, you must:

- Hold the INSERT privilege on or own the table, view, or table procedure named in the INTO parameter
- Hold the SELECT privilege on or own each table, function, view, and table procedure explicitly named in any query specification used in the INSERT statement

Additional authorization requirements apply to:

- A view named in the INTO parameter; each view named in the FROM parameter of such a view; each view named in the FROM parameters of those views, and so forth.
  
  For any such view, the owner of the view must own or have the grantable INSERT privilege on each table, view, and table procedure explicitly named in the view definition.

- Each view explicitly named in a query specification in the INSERT statement; each view explicitly named in the definition of such a view; each view explicitly named in the definition of those views, and so forth.
  
  For any such view, the owner of the view must own or have the grantable SELECT privilege on each table, view, and table procedure explicitly named in the view definition.

**Syntax**

```sql
INSERT INTO table-name (column-name) 
```
Expansion of bulk-options

- START : start-variable-name
- ROWS : row-count-variable-name
Parameters

**INTO table-name**

Identifies the table, view, or table procedure to which new rows are being added. Table-name must not specify a procedure. If `table-name` identifies a view:

- The view must be updateable
- The applicable rows are inserted into the table from which the view is derived

For expanded `table-name` syntax, see Expansion of Table-name.

**(column-name)**

Specifies one or more columns for which values are being supplied. `Column-name` must identify a column in the table, view, or table procedure named in the INTO parameter and must be unique within the list of columns.

The list of column names must be enclosed in parentheses. Multiple column names must be separated by commas.

If you specify one or more but not all the columns in the named table, view, or table procedure, CA IDMS stores default or null values in the unspecified columns. If any unspecified column is defined as NOT NULL and does not have a default value, CA IDMS returns an error.

If you do not specify any column names, you must supply values for all columns in the rows being added in the order in which the columns were specified in the table, view, or table procedure definition.

**VALUES**

Indicates a single row with the specified values is to be added to the table, view, or table procedure named in the INTO parameter. You must provide the same number of values as the number of columns named in the INSERT statement or, if no columns are named, the number of columns in the table, view, or table procedure. The first value specified is stored in the first column named, the second value in the second column, and so on.

The list of values must be enclosed in parentheses. Multiple values must be separated by commas.

**Important!** You can specify limited types of value expressions within a VALUES clause. For example, you cannot specify scalar functions or expressions with multiple terms.
NULL

Directs CA IDMS to store a null value in the corresponding column in the new row. The column must be defined to allow null values.

query-specification

Specifies a result table whose rows are to be added to the table, view, or table procedure named in the INTO parameter. The specified result table must have the same number of columns as the number of columns named in the INSERT statement or, if no columns are named, the number of columns in the table, view, or table procedure.

For expanded query-specification syntax, see Expansion of Query-specification.

BULK : bulk-buffer

Identifies a variable defined as an array from which CA IDMS is to retrieve the values to be stored in one or more new rows.

Bulk-buffer must be a variable previously declared in the host-language application program or SQL routine. Bulk-buffer must have a subordinate structure which occurs multiple times and has the same number of sub-elements as the number of columns named in the INSERT statement or, if no columns are named, the number of columns in the table, view, or table procedure named in the INTO parameter.

You can specify the BULK parameter only when you embed the INSERT statement in an application program.

The BULK parameter is a CA IDMS extension of the SQL standard.

Parameters for Expansion of bulk-options

START : start-variable-name

Identifies a variable containing the relative position within the bulk buffer from which CA IDMS is to retrieve values for the first new row. Values in subsequent entries in the bulk buffer are retrieved sequentially for subsequent new rows.

Start-variable-name must be a variable previously declared in the host-language application program or SQL routine. The value in the host variable must be an integer in the natural range of subscripts for arrays in the language in which the application program is written.

If you do not specify the START parameter, CA IDMS retrieves the values from the first entry in the bulk buffer.
ROWS : *row-count-variable-name*

Identifies a variable that specifies the number of rows CA IDMS is to retrieve from the bulk buffer.

*Row-count-variable-name* must be a variable previously declared in the host-language application program or SQL routine. The value in the host variable must be an integer in the range 1 through the number of rows that fit in the bulk buffer.

If you do not specify the ROWS parameter, CA IDMS retrieves rows from the array sequentially until reaching the end of the buffer.

**Note:** The *bulk-buffer, start-variable-name, row-count-variable-name* variables can be host variables, or when the statement is used in an SQL routine, local variables or routine parameters. In this case, their names must not be preceded with a colon.
### Usage

#### Restriction on table-name

The table, view, or table procedure named in the INTO parameter of an INSERT statement cannot also be named in the FROM parameter of a query specification in the same statement or in the FROM parameter of any subquery within the query specification. This means that you cannot INSERT into a table from which you are selecting directly or through a view.

#### Restriction for Tables in Constraints

If the table named in an INSERT statement is the referencing table in a constraint, the foreign-key columns in each row being inserted must satisfy either of the following conditions:

- The columns must be all or partially null
- The foreign-key-column values must match the referenced-column values in a row of the referenced table

#### Compatible Data Types

The data types of the columns named in the INSERT statement and their corresponding values represented by the VALUES, query-specification, or BULK parameter must be compatible for assignment.

#### Indicator Variables in the INSERT Statement

In an INSERT statement, you can use indicator variables with host variables and within arrays for bulk processing. A negative value in an indicator variable directs CA IDMS to store a null value in the column corresponding to the associated host variable or structure element. CA IDMS ignores an indicator value of 0 or higher.

#### Errors During Bulk Inserts

If an error occurs during a bulk insert, all rows inserted before the error occurred remain in the table. Subsequent rows, however, are not inserted into the table.

#### Satisfying Check Constraints

If a row to be inserted into a table does not satisfy the check constraints, if any, in the table definition, CA IDMS returns an error and does not insert the row.

#### Inserting into Views Having WITH CHECK OPTION

If the INTO parameter includes a view defined with WITH CHECK OPTION, any WHERE clause in the view definition, or in the definitions of any other views nested within its definition, is applied like a check constraint.
Examples

Supplying Explicit Values

The following INSERT statement adds a new row to the PROJECT table. Values (some null, some non-null) are provided for all columns in the table.

```sql
insert into project
values ('P634', 'TV ads - WTVK', '1989-12-01',
        '1990-2-28', null, null, 320, null, 3411);
```

Using the Values in a Result Table

The following INSERT statement adds new rows to the temporary table TEMP_BUDGET. The values in the new rows come from the result table defined by the query specification in the INSERT statement. Values are provided only for two columns of the temporary table.

```sql
insert into temp_budget
(dept_id, all_expenses)
select dept_id, adv_exp + merch_exp + op_exp + misc_exp
from proposed_budget;
```

Inserting Values from a Buffer

The following INSERT statement adds new rows to the CONSULTANT table. Values for the new rows come from the buffer CNSLT-BUFF. The number of rows added is determined by the value in the host variable BUF-ROW-COUNT.

```sql
EXEC SQL
INSERT INTO CONSULTANT
BULK :CNSLT-BUFF
ROWS :BUF-ROW-CNT
END-EXEC
```
OPEN

**More Information**

- For more information about updateable views, see CREATE VIEW.
- For more information about host variables, see Host Variables.
- For more information about expanded host variable, local variable, or routine parameters syntax, see Expansion of Host-variable, Local Variables (see page 81), or Routine Parameters.
- For more information about compatible data types for assignment operations, see Comparison, Assignment, Arithmetic, and Concatenation Operations.
- For more information about null values, see Null Values.
- For more information about bulk processing in an application program, see the CA IDMS SQL Programming Guide.
- For more information about expanded literal syntax, see Expansion of Literal.
- For more information about expanded special-register syntax, see Expansion of Special-register.

**OPEN**

The OPEN data manipulation statement places a specified cursor in the open state. An open cursor represents a result table, an ordering of the rows in the result table, and a position relative to the ordering.

You can use this statement only in SQL that is embedded in a program.

**Authorization**

None required.

**Syntax**

```
OPEN cursor-name

USING host-variable
  local-variable
  routine-parameter
  :dyn-buff sql DESCRIPTOR descriptor-area-name
```
Parameters

**cursor-name**

Specifies the cursor to be opened. **Cursor-name** must identify a cursor previously defined by a DECLARE CURSOR statement within the application program or by an ALLOCATE CURSOR statement executed within the same SQL transaction.

**USING**

Supplies values for the dynamic parameters embedded in the text of the dynamically prepared statement with which the cursor is associated.

**host-variable**

Identifies the host variables from which CA IDMS is to retrieve values for the dynamic parameters. CA IDMS assigns the value of the first host variable to the first dynamic parameter, the second host variable to the second dynamic parameter, and so on.

You must specify the same number of host variables in the USING parameter as the number of dynamic parameter markers in the dynamically prepared statement with which the cursor is associated. See Expansion of Host-variable, for more information.

**Note**: In COBOL, **host-variable** can be an elementary data item or a non-bulk structure. If a non-bulk structure is specified, each sub-element of the structure is counted as a host variable.

**local-variable**

**routine-parameter**

Identifies the local variable or routine parameter from which CA IDMS is to retrieve values for the dynamic parameters. CA IDMS assigns the value of the first local variable or routine parameter to the first dynamic parameter, the second local variable or routine parameter to the second dynamic parameter, and so on. You must specify the same number of local variables and routine parameters in the USING parameter as the number of dynamic parameter markers in the dynamically prepared statement with which the cursor is associated.

**:dyn-buff**

Identifies the variable from which CA IDMS is to retrieve values for the dynamic parameters.

**Dyn-buff** must identify a variable previously declared in the host-language application program or SQL routine.

The size of **dyn-buff** must be sufficient to hold a complete set of dynamic parameter values. The format of the data in **dyn-buff** must conform to the description in the SQL descriptor area specified by **descriptor-area-name**.
SQL DESCRIPTOR

Specifies the SQL descriptor area that describes the format of the dynamic parameter values contained in `dyn-buff`.

**descriptor-area-name**

Directs CA IDMS to use the named area as the descriptor area. `Descriptor-name` must identify an SQL descriptor area.

**Note:** The `dyn-buff` variable can be a host variable, or when the statement is used in an SQL routine, a local variable or a routine parameter. In this case, its name must not be preceded with a colon.

For the layout of an SQL descriptor area, see SQL Descriptive Area.

**Usage**

**Cursor Positions**

At any time, the position of an open cursor is one of the following:

- Before a certain row
- About a certain row
- After the last row

A cursor can be before the first row or after the last row of a result table even if the table is empty.

When a cursor is first opened, its position is before the first row.

**Effect on an Open Cursor**

If the cursor named in an OPEN statement is already open, CA IDMS returns an error and continues processing.

**Example**

**Opening a Cursor**

The following OPEN statement places the cursor named ALL_EMP_CURSOR in the open state:

```sql
EXEC SQL
   OPEN ALL_EMP_CURSOR
END-EXEC
```
More Information

- For more information about defining and manipulating cursors, see ALLOCATE CURSOR, CLOSE, DECLARE CURSOR, and FETCH (see page 449).
- For more information about using cursors in an application program, see the CA IDMS SQL Programming Guide.
- For more information about the layout of the descriptor area, see SQL Descriptor Area.

PREPARE

The PREPARE dynamic compilation statement dynamically compiles an SQL statement for later execution in the application program.

You can use this statement only in SQL that is embedded in a program.

Authorization

To issue the PREPARE statement, you must have the privileges required to issue the statement being prepared.

Syntax

```
PREPARE statement-name FROM :statement-text USING sql DESCRIPTOR descriptor-area-name1
```

Expansion of `describe-output-expression`

```
DESCRIBE output USING sql DESCRIPTOR descriptor-area-name1
```

Expansion of `describe-input-expression`

```
DESCRIBE INPUT USING sql DESCRIPTOR descriptor-area-name2
```

```
OUTPUT USING sql DESCRIPTOR descriptor-area-name1
```
**Parameters**

**statement-name**

Specifies the name to be assigned to the compiled statement. It must be unique within its associated scope.

**Note:** For more information, see Expansion of Statement-name.

**FROM**

Identifies the statement to be compiled.

**statement-text**

Identifies a host variable, local variable, or a routine parameter containing a preparable SQL statement. `statement-text` must be previously declared in the application program or SQL routine. It must be defined as an elementary data item with no sub-elements. Do not specify the colon when `statement-text` is a local variable or routine parameter.

```
'statement-text'
```

Specifies a preparable SQL statement enclosed in single quotation marks. Do not include the SQL prefix or terminator within the statement.

**Parameters for Expansion of describe-output-expression**

**DESCRIBE OUTPUT USING SQL DESCRIPTOR descriptor-area-name1**

Specifies the SQL descriptor area in which CA IDMS is to return information about the output values to be returned when the dynamically-compiled statement is executed. `Descriptor-area-name1` is the name of the SQL descriptor area.

**INPUT USING SQL DESCRIPTOR descriptor-area-name2**

Specifies the SQL descriptor area in which CA IDMS is to return information about the dynamic parameters used within the statement. `Descriptor-area-name2` is the name of the SQL descriptor area.

**Parameters for Expansion of describe-input-expression**

**DESCRIBE INPUT USING SQL DESCRIPTOR descriptor-area-name2**

Specifies the SQL descriptor area in which CA IDMS is to return information about the dynamic parameters used within the statement. `Descriptor-area-name2` is the name of the SQL descriptor area.

**OUTPUT USING SQL DESCRIPTOR descriptor-area-name1**

Specifies the SQL descriptor area in which CA IDMS is to return information about the output values to be returned when the dynamically-compiled statement is executed. `Descriptor-area-name1` is the name of the SQL descriptor area.
Usage

Preparable Statements

The following SQL statements are preparable:

- All authorization and logical data description statements
- CALL
- COMMIT
- cursor-specification
- DELETE
- EXPLAIN
- INSERT
- RELEASE
- ROLLBACK
- SUSPEND SESSION
- UPDATE

Additionally, all CA IDMS utility and physical data description statements are preparable.

Specifying Dynamic Parameters

Dynamic parameters are variables whose values are supplied when the statement is executed, or in the case of a SELECT or a CALL statement, when its associated cursor is opened.

Dynamic parameters are specified as question marks (?) within the text of the SQL statement. They may appear wherever a host variable is permitted with certain exceptions.

Describing Dynamic Parameters

The INPUT option is used to return information about dynamic parameters that may be embedded in the SQL statement being described. The SQLD field of the descriptor area indicates the number of dynamic parameter that appear in the statement. If no dynamic parameters are used, this field is zero (0).

If dynamic parameters do appear in the statement, CA IDMS returns descriptions of the parameters in the descriptor area. The data type information is derived from the context in which the dynamic parameter appears.
Describing Output Values

The OUTPUT option is used to return information about values output from CA IDMS:

- For a SELECT or a CALL statement, CA IDMS returns a description of the result table defined by the statement. The SQLD field of the descriptor area indicates the number of columns in the result table.
- For a statement other than SELECT or CALL, CA IDMS returns the value zero (0) in the SQLD field of the descriptor area.

No Host Variables, Local Variables, or Routine Parameters in a Dynamically Compiled Statement

An SQL statement that is to be compiled dynamically cannot include any host variables, local variables, or routine parameters.

Re-executing a PREPARE Statement

When reexecuting a PREPARE statement, CA IDMS replaces the previously prepared statement with the statement currently identified in the PREPARE statement. If the previously prepared statement is a SELECT or a CALL statement associated with an open cursor, CA IDMS closes the cursor.

Duration of Dynamically Compiled Statements

Dynamically-compiled statements are available for execution until the transaction terminates or until destroyed using a DEALLOCATE PREPARE statement.

Specifying the Maximum Number of Column Entries

The application program must specify the maximum number of entries it can accept by setting the value of the SQLN field in the descriptor area before issuing the PREPARE statement. If the number of entries is insufficient, CA IDMS returns the number of entries needed into the SQLD field but does not return any descriptions.
Examples

Specifying the Statement Explicitly

The following PREPARE statement dynamically compiles the specified SELECT statement. A subsequent DESCRIBE statement must provide a descriptor area for the description of the result table before the dynamically compiled statement can be executed.

EXEC SQL
  PREPARE DYN_TMP_SEL_1
    FROM 'SELECT * FROM TEMP_BUDGET'
END-EXEC

Using a Host Variable

The following PREPARE statement dynamically compiles the statement contained in the host variable SELECT-BUFF. Information about the output from the dynamically compiled statement is returned in the descriptor area named BUFF-1-SQLDA.

EXEC SQL
  PREPARE DYN_PROJ_SELECT
    FROM :SELECT-BUFF
  DESCRIBE USING DESCRIPTOR BUFF-1-SQLDA
END-EXEC

More Information

- For more information about the dynamic compilation of SQL statements, see the CA IDMS SQL Programming Guide.
- For more information about the SQL descriptor area, see SQL Descriptor Area.
- For more information about destroying a dynamically-compiled statement, see DEALLOCATE PREPARE.
- For more information about the structure of the SQL descriptor area, see SQL Descriptor Area.
- For more information about dynamic parameters, see Dynamic Parameters.

RELEASE

The RELEASE session management statement releases a connection to a CA IDMS dictionary and ends the SQL session. It is also a CA IDMS extension of the SQL standard.

Authorization

None required.
Syntax

RELEASE

Usage

Ending an SQL Session

To end an SQL session established with the CONNECT statement, you must use one of the following statements:

- RELEASE
- COMMIT RELEASE
- ROLLBACK RELEASE

To end an SQL Session established automatically, you can use any of the above statements or COMMIT or ROLLBACK without the RELEASE parameter.

Automatic Rollback

When ending an SQL session, CA IDMS automatically rolls back any transaction that is still active.

Example

Releasing a Connection

The following RELEASE statement ends the current SQL session and releases the connection with the dictionary:

EXEC SQL
    RELEASE
END-EXEC

More Information

- For more information about establishing a database connection and beginning an SQL session, see CONNECT.
- For more information about managing SQL sessions, see the CA IDMS SQL Programming Guide.

RESUME SESSION

The RESUME SESSION management statement resumes a suspended SQL session. You use the RESUME SESSION statement primarily in pseudo conversational programming. It is also a CA IDMS extension of the SQL standard.
Authorization

None required.

Syntax

**RESUME SESSION**

Usage

**Effect of RESUME SESSION**

When you resume an SQL session, CA IDMS re-establishes the session as it existed immediately before it was suspended.

**Resuming a Suspended Session**

The following RESUME SESSION statement resumes the current suspended SQL session:

```
EXEC SQL
RESUME SESSION
END-EXEC
```

More Information

- For more information about suspending an SQL session, see SUSPEND SESSION.
- For more information about managing SQL sessions, see the *CA IDMS SQL Programming Guide*.

**REVOKE All Table Privileges**

The REVOKE All Table Privileges authorization statement removes from one or more users or groups all definition and access privileges on a specified table, view, function, procedure or table procedure.

Authorization

To issue the REVOKE ALL PRIVILEGES statement, one of the following must be true:

- You own the table, view, function, procedure or table procedure
- You hold all privileges on the table, view, function, procedure or table procedure as grantable
- You hold the DBADMIN privilege for the database that contains the table, view, function, procedure or table procedure
REVOKE All Table Privileges

Syntax

```
REVOKE ALL PRIVILEGES
ON table table-name
FROM PUBLIC
```

Parameters

**ALL PRIVILEGES**

Removes the DELETE, INSERT, SELECT, UPDATE, ALTER, CREATE, DROP, and REFERENCES privileges, as applicable, on the table, view, function, procedure, or table procedure identified in the ON parameter from the users or groups identified in the FROM parameter.

**ON table table-name**

Identifies the table, view, function, procedure or table procedure to which the privileges apply.

If `table-name` does not include a schema name qualifier, the schema name qualifier defaults to the current schema in effect for your SQL session. For expanded `table-name` syntax, see Expansion of Table-name.

**schema-name**

Specifies the schema with which the function identified by `function-identifier` is associated. If `schema-name` is not specified, the schema qualifier defaults to the current schema in effect for your SQL session.

**function-identifier**

Identifies the function to which the privileges apply.

**FROM**

Specifies the users from whom you are removing the privileges.

**PUBLIC**

Specifies all users.

The privileges must have been previously given to PUBLIC by means of the GRANT statement.

**authorization-identifier**

Identifies a user or group.

The privileges must have been previously given to `authorization-identifier` by means of the GRANT statement. For expanded `authorization-identifier` syntax, see Expansion of Authorization-identifier.
Usage

Revoking Privileges

A user can hold a privilege on a resource through multiple resource names (for example, through the use of wildcards) or through multiple authorization identifiers (for example, through two different group identifiers). A REVOKE statement revokes the privileges specified in the statement only on the specified resource name and only from the specified authorization identifier.

For example, suppose:

- User PKB is in the group SALES_ADMIN
- PKB has been granted the SELECT privilege on the table SALES_SCH.SALES_FORECAST
- SALES_ADMIN has been granted the SELECT privilege on all tables named SALES_SCH.SALES* where * is a wildcard character

You can revoke the SELECT privilege on SALES_FORECAST from the user identifier PKB. However, PKB can still select data from the table because PKB is a member of SALES_ADMIN.

Example

Revoking All Privileges From All Users

The following statement removes all privileges on all tables, views, functions, procedures and table procedures in the TEST schema from the group PUBLIC:

```
revoke all privileges
  on test.*
  from public;
```

More Information

- For more information about table access privileges, see GRANT Table Access Privileges and REVOKE Table Access Privileges.
- For more information about table definition privileges, see GRANT Definition Privileges and REVOKE SQL Definition Privileges.
- For more information about granting privileges, see your security administrator.
REVOKE SQL Definition Privileges

The REVOKE SQL Definition Privileges authorization statement removes from one or more users or groups the privilege of performing selected actions on a specified access module, schema, table, view, function, procedure or table procedure. It is also a CA IDMS extension of the SQL standard.

Authorization

To issue a REVOKE statement for a definition privilege, you must own the resource, hold the corresponding grantable privilege on the resource, or hold DBADMIN privilege on the dictionary containing the definition.

Syntax

```
REVOKE [DEFINE | ALTER | CREATE | DISPLAY | DROP | REFERENCES] ON ACCESS MODULE [schema-name.] access-module-name
  [SCHEMA schema-name]
  [table table-name]
  [function function-identifier]
FROM [PUBLIC | authorization-identifier]
```
Parameters

DEFINE
Removes the ALTER, CREATE, DISPLAY, DROP, and REFERENCES privileges, as applicable, on the resource identified in the ON parameter from the users or groups identified in the FROM parameter.

ALTER
Removes the ALTER privilege on the resource identified in the ON parameter from the users or groups identified in the FROM parameter.

CREATE
Removes the CREATE privilege on the resource identified in the ON parameter from the users or groups identified in the FROM parameter.

DISPLAY
Removes the DISPLAY privilege on the resource identified in the ON parameter from the users or groups identified in the FROM parameter.

DROP
Removes the DROP privilege on the resource identified in the ON parameter from the users or groups identified in the FROM parameter.

REFERENCES
Removes the REFERENCES privilege on the resource identified in the ON parameter from the users or groups identified in the FROM parameter.

ON
Identifies the resource to which the named privileges apply.

ACCESS MODULE access-module-name
Identifies an access module.
Privileges on any version of access-module-name in the associated schema are revoked.

schema-name
Identifies the schema associated with the named access module.
If you do not specify schema-name, it defaults to the current schema in effect for your SQL session.

SCHEMA schema-name
Identifies a schema.

table table-name
Identifies a table, view, procedure or table procedure.
If table-name does not include a schema name qualifier, the schema name qualifier defaults to the current schema in effect for your SQL session.
REVOKE SQL Definition Privileges

schema-name

Specifies the schema with which the function identified by function-identifier is associated. If schema-name is not specified, the schema defaults to the current schema in effect for your SQL session.

function-identifier

Identifies the function.

FROM

Identifies the users from whom you are removing the specified privileges.

PUBLIC

Specifies all users.

The privileges must have been previously given to PUBLIC by means of the GRANT statement.

authorization-identifier

Identifies a user or group.

The privileges must have been previously given to authorization-identifier by means of the GRANT statement. For expanded authorization-identifier syntax, see Expansion of Authorization-identifier.
### Usage

#### Revoking Privileges

A user can hold a privilege on a resource through multiple resource names (for example, through the use of wildcards) or through multiple authorization identifiers (for example, through two different group identifiers). A REVOKE statement revokes the privileges specified in the statement only on the specified resource name and only from the specified authorization identifier.

For example, suppose:

- User PKB is in the group SALES_ADMIN
- PKB has been granted the CREATE privilege on the access module name SALES_SCH.SALESFCT
- SALES_ADMIN has been granted the CREATE privilege on all access modules named SALES_SCH.SALES* where * is a wildcard character

You can revoke the CREATE privilege on SALESFCT from the user identifier PKB. However, PKB can still create an access module by that name in the SALES_SCH schema because PKB is a member of SALES_ADMIN.

#### The DEFINE Keyword

When you use the DEFINE keyword with a GRANT statement, you grant a set of definition privileges on a resource to one or more users or groups.

When you use the DEFINE keyword with a REVOKE statement, you revoke any definition privileges that have been previously granted on the resource from the specified users or groups.

This means that if you GRANT CREATE privilege on a table, you can revoke the privilege with a REVOKE SELECT statement or a REVOKE DEFINE statement. Using REVOKE DEFINE is an efficient technique when you intend to revoke all definition privileges on a table from a user or group, whether the privileges were granted singly or as a set.

Similarly, you can GRANT DEFINE on a table to a user and then REVOKE DELETE on the table from the same user as a way to grant all but one definition privilege.
Example

**Revoking Privileges on a Schema**

The following REVOKE statement removes the ALTER, CREATE, DISPLAY, and DROP privileges on all schemas that begin with 'DSF' from user DSF:

```
revoke define
   on schema dsf*
from dsf;
```

More Information

- For more information about granting definition privileges, see GRANT Definition Privileges.
- For more information about revoking privileges, see your security administrator.

**REVOKE Execution Privilege**

The REVOKE Execution Privilege authorization statement removes from one or more users or groups the privilege of executing a specified access module. It is also a CA IDMS extension of the SQL standard.

**Authorization**

To revoke access module execution privilege, you must own the schema associated with the access module, hold grantable privilege on the access module, or hold DBADMIN privilege on the dictionary that contains the access module.

**Syntax**

```
REVOKE EXECUTE
ON ACCESS MODULE schema-name. access-module-name
FROM PUBLIC authorization-identifier
```
Parameters

ON ACCESS MODULE access-module-name

   Specifies the access module to which the EXECUTE privilege applies.

   schema-name

      Identifies the schema associated with access-module-name. If you do not
      specify schema-name, it defaults to the current schema in effect for your SQL
      session.

FROM

   Identifies the users from whom you are removing the EXECUTE privilege.

PUBLIC

   Specifies all users.

   The privilege must previously have been granted to PUBLIC.

authorization-identifier

   Identifies a user or group.

   The privilege must previously have been granted to authorization-identifier. For
   expanded authorization-identifier syntax, see Expansion of Authorization-identifier.

Usage

Revoking Privileges

A user can hold a privilege on a resource through multiple resource names (for example,
through the use of wildcards) or through multiple authorization identifiers (for example,
through two different group identifiers). A REVOKE statement revokes the privileges
specified in the statement only on the specified resource name and only from the
specified authorization identifier.

For example, suppose:

■ User PKB is in the group SALES_ADMIN
■ PKB has been granted the EXECUTE privilege on the access module name
  SALES_SCH.SALESFCT
■ SALES_ADMIN has been granted the EXECUTE privilege on all access module named
  SALES_SCH.SALES* where * is a wildcard character

You can revoke the EXECUTE privilege on SALESFCT from the user identifier PKB.
However, PKB can still execute an access module by that name in the SALES_SCH
schema because PKB is a member of SALES_ADMIN.
**Example**

**Revoking the EXECUTE Privilege**

The following REVOKE EXECUTE statement removes the EXECUTE privilege on all access modules associated with schema HR that begin with 'EMP' from the users in group PER_GRP_2:

```sql
revoke execute
  on access module hr.emp*
from per_grp_2;
```

**More Information**

- For more information about granting the EXECUTE privilege, see GRANT Access Module Execution Privilege.
- For more information about revoking privileges, see your security administrator.

**REVOKE Table Access Privileges**

The REVOKE Table Access Privileges authorization statement removes from one or more users or groups the privilege of performing selected actions on a specified table, view, function, procedure or table procedure.

**Authorization**

To issue a REVOKE statement for a table, view, function, procedure or table procedure privilege, you must own, hold the corresponding grantable privilege on the table, view, function, procedure or table procedure, or hold the DBADMIN privilege on the database.

**Syntax**

```
REVOKE ACCESS DELETE
  INSERT SELECT UPDATE
ON table table-name
  function-identifier
FROM PUBLIC authorization-identifier
```

REVOKE Table Access Privileges

Parameters

ACCESS
Removes the DELETE, INSERT, SELECT, and UPDATE privileges on the named table, view, function, procedure or table procedure from the users or groups identified in the FROM parameter.

DELETE
Removes the DELETE privilege on the table, view, or table procedure identified in the ON parameter from the users or groups identified in the FROM parameter.

INSERT
Removes the INSERT privilege on the table, view, or table procedure identified in the ON parameter from the users or groups identified in the FROM parameter.

SELECT
Removes the SELECT privilege on the table, view, function, procedure or table procedure identified in the ON parameter from the users or groups identified in the FROM parameter.

UPDATE
Removes the UPDATE privilege on the table, view, or table procedure identified in the ON parameter from the users or groups identified in the FROM parameter.

ON table table-name
Specifies the table, view, procedure or table procedure which the access privileges apply.

If table-name does not include a schema name qualifier, the schema name qualifier defaults to the current schema in effect for your SQL session. For expanded table-name syntax, see Expansion of Table-name.

schema-name
Specifies the schema with which the function identified by function-identifier is associated. If schema-name is not specified, the schema qualifier defaults to the current schema in effect for your SQL session.

function-identifier
Identifies the function to which the privileges apply.

FROM
Identifies the users from whom you are removing access privileges.
PUBLIC

Specifies all users.

The privileges must have been previously given to PUBLIC by means of the GRANT statement.

authorization-identifier

Identifies a user or group.

The privileges must have been previously given to authorization-identifier by means of the GRANT statement. For expanded authorization-identifier syntax, see Expansion of Authorization-identifier.
Usage

The ACCESS Keyword

When you use the ACCESS keyword with a GRANT statement, you grant a set of access privileges on a table, view, function, procedure or table procedure to one or more users or groups.

When you use the ACCESS keyword with a REVOKE statement, you revoke any access privileges that have been previously granted on the table, view, function, procedure or table procedure from the specified users or groups.

Therefore, if you GRANT SELECT privilege on a table, you can revoke the privilege with a REVOKE SELECT statement or a REVOKE ACCESS statement. Using REVOKE ACCESS is an efficient technique when you intend to revoke all access privileges on a table from a user or group, whether the privileges were granted singly or as a set.

Similarly, you can GRANT ACCESS on a table to a user and then REVOKE DELETE on the table from the same user as a way to grant all but one table access privilege.

Revoking Privileges

A user can hold a privilege on a resource through multiple resource names (for example, through the use of wildcards) or through multiple authorization identifiers (for example, through two different group identifiers). A REVOKE statement revokes the privileges specified in the statement only on the specified resource name and only from the specified authorization identifier.

For example, suppose:

- User PKB is in the group SALES_ADMIN
- PKB has been granted the SELECT privilege on the table name SALES_SCH.SALES_FORECAST
- SALES_ADMIN has been granted the SELECT privilege on all tables named SALES_SCH.SALES* where * is a wildcard character

You can revoke the SELECT privilege on SALES_FORECAST from the user identifier PKB. However, PKB can still select from the SALES_FORECAST table because PKB is a member of SALES_ADMIN.
Example

Revoking Selected Privileges on a Table

The following REVOKE statement removes the SELECT and UPDATE privileges on the EMPLOYEE table from users KRP, SAE, and PGD:

revoke select, update
    on employee
    from krp, sae, pgd;

More Information

- For more information about granting table access privileges, see GRANT Table Access Privileges.
- For more information about revoking privileges, see your security administrator.

ROLLBACK

The ROLLBACK transaction management statement performs the following tasks:

- Cancels changes made to the database during the current transaction
- Ends the transaction
- Optionally ends the SQL session

Authorization

None required.

Syntax

```
ROLLBACK work
  RELEASE
```

Parameters

RELEASE

Directs CA IDMS to end the current SQL session and the current transaction after canceling the changes to the database.

The RELEASE parameter is a CA IDMS extension of the SQL standard.
Usage

Effect of a ROLLBACK on an SQL Session

A ROLLBACK statement has the following impact on the SQL session and its transaction:

- Rolls back all changes made by the session
- Releases all locks
- Closes all open cursors
- Drops all temporary tables
- Deletes all dynamically compiled statements
- Terminates the SQL session CA IDMS connected it automatically or if RELEASE is specified.

Effect of Transaction Sharing

If more than one database session is sharing the SQL session’s transaction, the changes made by all sharing sessions are immediately rolled back. All sharing sessions other than the one through which the ROLLBACK statement was issued are flagged to indicate that they must also issue a ROLLBACK. If the next statement issued by each of these sessions is not a ROLLBACK, it will receive an error:

- For SQL, the application receives an SQLCODE of -5 (transaction failure) and an SQLRSN of 1088 (transaction forced to backout).
- For navigational DML, the run unit is terminated and an error status of xx19 is returned to the application.

Example

Canceling Database Changes

The following ROLLBACK statement cancels the uncommitted changes to the database made during the current transaction and ends both the transaction and the current SQL session:

```sql
EXEC SQL
   ROLLBACK RELEASE
END-EXEC
```

More Information

- For more information about committing changes to the database before ending a transaction, see COMMIT.
- For more information about ending an SQL session, see RELEASE.
- For more information about managing or sharing transactions, see the CA IDMS SQL Programming Guide.
SELECT

The SELECT data manipulation statement retrieves values from one or more tables, views, procedures and table procedures. CA IDMS returns the values in the form of a result table.

When the SELECT statement is:
- Submitted through the Command Facility, the values in the result table are displayed in tabular form
- Embedded in an application program or SQL routine, the values in the result table are stored in host variables, local variables, or routine parameters

Authorization

To issue a SELECT statement, you must own or have the SELECT privilege on each table, view, function, procedure and table procedure explicitly named in the statement.

Additional authorization requirements apply to each view explicitly named in the SELECT statement, to each view explicitly named in the definition of such a view, to each view explicitly named in the definition of those views, and so forth.

For any such view, the owner of the view must own or have the grantable SELECT privilege on each table, view, procedure and table procedure explicitly named in the view definition.

Syntax
Parameters

ALL

Directs CA IDMS to return all the rows, including duplicates, in the requested result table. The default value is ALL when you specify neither ALL nor DISTINCT.

DISTINCT

Directs CA IDMS to eliminate duplicate rows from the result table returned by the SELECT statement.

*  

Specifies that the result table is to include all columns in the tables, views, procedures and table procedures named in the FROM parameter of the SELECT statement. The columns in the tables, views, procedures and table procedures are concatenated in the order in which the tables, views, procedures and table procedures are specified in the FROM parameter.
value-expression

Identifies the values to be included in a result column. Typically, value-expression is a column reference, an arithmetic operation that includes a column reference, or an aggregate function that includes a column reference.

Each column reference in value-expression must identify a column in the table defined by the FROM parameter of the SELECT statement.

You can specify from 1 through 1,024 value expressions. Multiple value expressions must be separated by commas.

The number of columns in a result table is the same as the number of value expressions in the SELECT statement defining the result table. For expanded value-expression syntax, see Expansion of Value-expression.

AS result-name

Specifies a name for the result column identified by value-expression. When displaying the result table, the Command Facility uses the result name as the column header.

Result-name must be a 1- through 32-character name that follows the conventions for SQL identifiers.
table-name.*

Specifies that the result table is to include all columns in the table identified by table-name.

Table-name must match an occurrence of table-name in the FROM parameter.

alias.*

Specifies that the result table is to include all columns in the table identified by alias.

Alias must match an occurrence of alias in the FROM parameter.

INTO host-variable
local-variable
routine-parameter

Identifies the variables to which CA IDMS is to assign the values in the result table. CA IDMS assigns the value in the first result column to the first variable, the value in the second result column to the second variable, and so on. You use the INTO parameter when the result table will contain at most one row.

Host-variable must be a host variable previously declared in the application program.

Local-variable and routine-parameter must be defined previously in the SQL routine.

You must specify the same number of variables in the INTO parameter as the number of columns in the result table. Multiple variables must be separated by commas.

You can specify the INTO parameter only when you embed the SELECT statement in an application program or SQL routine. You must specify INTO or BULK when you embed a SELECT statement in a host program or SQL routine.
BULK : bulk-buffer

Identifies a variable defined as an array to which CA IDMS is to assign the values in the result table. The BULK parameter is a CA IDMS extension of the SQL standard. You use the BULK parameter when the result table may contain more than one row.

You can specify the BULK parameter only when you embed the SELECT statement in an application program. You must specify BULK or INTO when you embed a SELECT statement in a host program.

Bulk-buffer must be a variable previously declared in the host-language application program or SQL routine. Bulk-buffer must have a subordinate structure that occurs multiple times and has the same number of sub-elements as the number of columns in the result table.

bulk-options

Refers to optional parameters when BULK is specified. Syntax for bulk-options immediately follows the syntax for SELECT.

FROM table-reference

Identifies one or more tables, views, procedures and table procedures from which the result table is to be derived. For expanded table-reference syntax, see Expansion of Table-reference.

(query-expression)

Represents a table to be used in the evaluation of an SQL statement.

AS alias

Defines a new name to be used to identify the table, view, procedure, table procedure or query-expression within the SELECT statement. Alias must be a 1-through 18-character name that follows the conventions for SQL identifiers.

Note: CA IDMS supports keywords as identifiers as an extension of the SQL standard. However, if you use a keyword as an alias but do not code the optional parameter AS, you must delimit the keyword with double quotation marks or a syntax error will occur.
WHERE

Introduces criteria that a row must meet to be included in the result table.

search-condition

Specifies the set of values against which a row in the base table is tested:

- When the value of search-condition is true, the row is included in the result table
- When the value of search-condition is false or unknown, the row is not included in the result table

For expanded search-condition syntax, see Expansion of Search-condition.

extended-search-condition

Specifies a search condition that includes a set specification. For expanded extended-search-condition syntax, see Expansion of Extended-search Condition.

PRESERVE

Requests an outer join on the specified table, view, procedure, or table procedure. The PRESERVE parameter is a CA IDMS extension of the SQL standard.

To specify a more powerful outer join that is compatible with the SQL standard, use the joined-table construct as table-reference

table-name

Specifies by table name the table, view, procedure or table procedure to be preserved in an outer join. For expanded table-name syntax, see Expansion of Table-name.
**SELECT**

*alias*

Specifies the table, view, procedure or table procedure to be preserved in an outer join by the alias defined for the table or view in the FROM parameter of the SELECT statement.

**GROUP BY column-name**

Groups the rows in the table defined by the FROM parameter by the values in the specified columns. Rows with the same value in each grouping column are grouped together.

*Column-name* must identify a column in a table, view, procedure or table procedure named in the FROM parameter of the SELECT statement.

*table-name*

Specifies the table, view, procedure or table procedure that includes the named column. For expanded *table-name* syntax, see Expansion of Table-name.

*alias*

Specifies the alias associated with the table, view, procedure or table procedure that includes the named column. *Alias* must be defined in the FROM parameter of the SELECT statement.

*rowid-pseudo-column*

Specifies a ROWID pseudo-column as a grouping column. See Expansion of rowid-pseudo-column for more information.

**HAVING search-condition**

Specifies criteria a group must meet to be included in the result table:

- When the value of *search-condition* is true, the group is included in the result table
- When the value of *search-condition* is false or unknown, the group is not included in the result table

For expanded *search-condition* syntax, see Expansion of Search-condition.
OPTIMIZE FOR literal ROWS

Specifies the expected number of output rows from this query-specification. It is used by the optimizer to generate the best possible access strategy for satisfying query-expression. "Literal" is an integer constant.

UNION query-expression

Specifies that:

■ The result table is to include both the rows from the table defined in the FROM parameter of the SELECT statement and the rows from the table defined in query-expression.

■ Duplicate rows are to be eliminated from the table resulting from the UNION operation, unless the ALL keyword is present.

You cannot include the UNION operator in a SELECT statement embedded in an application program.

See Expansion of Query-expression for:

■ Expanded query-expression syntax

■ A discussion of data type compatibility and the data type that results from the union of columns with compatible data types

ALL

Specifies that all rows resulting from the UNION operation are retained; duplicates are not discarded.

ORDER BY

Sorts the rows in the table defined by the FROM parameter in ascending or descending order by the values in the specified columns. Rows are ordered first by the first column specified, then by the second column specified within the ordering established by the first column, then by the third column specified, and so on.

column-name

Specifies a sort column by the column name. Column-name must identify a column in a table, view, procedure or table procedure named in the FROM parameter of the SELECT statement and must be included in the result table.

table-name

Specifies the table, view, procedure or table procedure that includes the named column. For expanded table-name syntax, see Expansion of Table-name.

alias

Specifies the alias associated with the table, view, procedure or table procedure that includes the named column. Alias must be defined in the FROM parameter of the SELECT statement.
column-number

Specifies a sort column by the position of the column in the result table. The first result column is in position 1.

Column-number must be an integer in the range 1 through the number of columns in the result table.

result-name

Specifies the sort column by the result name specified in the AS parameter of query-expression.

rowid-pseudo-column

Specifies a sort column as a ROWID pseudo-column. See Expansion of rowid-pseudo-column for more information.

ASC

Indicates that the values in the specified column are to be sorted in ascending order. ASC is the default when you specify neither ASC nor DESC.
**DESC**

Indicates that the values in the specified column are to be sorted in descending order.

**Parameters for Expansion of bulk-options**

**START : start-variable-name**

Identifies a variable containing the relative position within the bulk buffer to which CA IDMS is to assign the values in the first row of the result table. Values in subsequent rows of the result table are assigned sequentially to subsequent positions in the bulk buffer.

`Start-variable-name` must be a variable previously declared in the host-language application program or SQL routine. The value in the variable must be an integer in the natural range of subscripts for arrays in the language in which the application program is written.

If you do not specify the START parameter, CA IDMS assigns the values in the first row of the result table to the beginning of the bulk buffer.

**ROWS : row-count-variable-name**

Identifies a variable that specifies the maximum number of rows in the result table CA IDMS is to assign to the bulk buffer.

`Row-count-variable-name` must be a variable previously declared in the host-language application program or SQL routine. The value in the host variable must be an integer in the range 1 through the number of rows that will fit in the bulk buffer.

If you do not specify the ROWS parameter, CA IDMS assigns the rows in the result table to the bulk buffer sequentially until no more rows exist in the result table or the buffer is full.

**Note:** The bulk-buffer, start-variable-name, and row-count-variable-name variables can be host variables, or when the statement is used in an SQL routine, local variables or routine parameters. In this case, their names must not be preceded with a colon.
Usage

Value Expressions without Column References

If the value expression that identifies a result column does not include any column references, the result column contains the same value in each row. This value is derived directly from the value expression without reference to the table defined by the FROM parameter of the SELECT statement.

Use of BULK and INTO

You must specify the BULK parameter or the INTO parameter when you embed the SELECT statement in an application program, except when the statement is to be compiled dynamically.

You cannot specify either of these parameters when you submit the SELECT statement through the command facility or for dynamic compilation in an application program.

When you embed the SELECT statement in an application program and:

- You specify INTO, the result table must have at most one row
- You specify BULK, the result table must have no more rows than the number of entries in the bulk buffer (or the value of row-count-variable-name, if specified)
If neither of these conditions is met, CA IDMS returns a cardinality violation error.

**Note:** To select an undetermined number of rows, the SELECT statement must be associated with a cursor. You can fetch rows individually from the cursor.

**Compatible Data Types**

The data types of the result columns and their corresponding host variables in the BULK or INTO parameter must be compatible for assignment.

**Uniqueness of Table Names**

Each alias and each table name without an associated alias must be unique within the FROM parameter of a SELECT statement.

**Column References in the WHERE Parameter**

Each column reference directly included in the search condition in the WHERE parameter of a SELECT statement must identify a column in a table, view, procedure or table procedure specified in the FROM parameter of the SELECT statement, or be an outer reference.

**Note:** For more information about outer references, see Subqueries.

**Aggregate Functions in the WHERE Parameter**

The search condition in the WHERE parameter of a SELECT statement cannot directly include an aggregate function. However, you can use aggregate functions in subqueries within the search condition.

**GROUP BY Parameter Requirements**

When a SELECT statement includes the GROUP BY parameter, each column reference in the value expressions that identify the result columns must identify a column specified in the GROUP BY parameter or occur only in the argument of an aggregate function. If the result columns are identified by an asterisk (*), the GROUP BY parameter must include all the columns in the tables, views, procedures and table procedures specified in the FROM parameter.

**SELECT Statements without the GROUP BY Parameter**

If a SELECT statement does not include the GROUP BY parameter:

- If any column reference in a value expression that identifies a result column is included in the argument of an aggregate function, all column references in all the value expressions must be in aggregate functions
- The entire table defined by the FROM and WHERE parameters is treated as a single group
Column References in the HAVING Parameter

Each column reference included in the search condition in the HAVING parameter of a SELECT statement must identify a column specified in the GROUP BY parameter of the SELECT statement or occur in the argument of an aggregate function.

When to Use OPTIMIZE FOR Literal ROWS

Under some circumstances, the SQL optimizer may choose a less than optimal access strategy to satisfy a query expression. This typically happens with host program embedded SQL statements which contain WHERE clauses with host variable references, rather than explicit constants. For example, a BETWEEN clause involving host variables may induce the optimizer to assume many rows will be retrieved, causing it to choose an area sweep to satisfy the request. Without knowing the underlying values of the host variables, the optimizer cannot know if the BETWEEN will always qualify a small number of rows, thus possibly making an index retrieval much more efficient. The OPTIMIZE FOR literal ROWS clause is used to override the number of expected rows deduced by the optimizer. This allows it to generate better access strategies.
**Result Column Names with the UNION Operator**

When a SELECT statement includes the UNION operator, the names of the columns in the result table are the names established by the last UNION operand. These names are used as:

- Column headings when the online command facility displays the result table
- Column names in the SQL descriptor area when CA IDMS compiles the SELECT statement dynamically

**Outer Join Using PRESERVE**

Within a SELECT statement, PRESERVE can be used to request an outer join on one of the tables or views named in the FROM parameter. If PRESERVE is specified, the result table includes rows of the preserved table for which no matching row exists in the other tables used in the join operation.

If no matching row exists, the corresponding columns in the result table are set to null. Predicates in the WHERE clause other than those used to perform the outer join are evaluated before determining whether a matching row exists.

The following statement returns the names of all active employees. The name of the employee's spouse is also returned if found. The logic of the statement is that the result table will include the name of each active employee, whether the employee has a spouse:

```sql
select e.first_name, e.last_name, 
     s.first_name, s.last_name 
from employee e, relation s 
where e.empid=s.empid 
   and e.status='A'  -- active employee 
   and s.relationship='S'  -- employee's spouse 
preserve e ;
```
Examples

Selecting a Single Row

The following SELECT statement retrieves information about a specific project from the PROJECT and EMPLOYEE tables. The value in each selected column is assigned to the corresponding host variable. The SELECT statement includes indicator variables for the ACT_START_DATE, ACT_END_DATE, EST_START_DATE, and EST_END_DATE columns.

```
EXEC SQL
  SELECT PROJ_ID, EMP_FNAME, EMP_LNAME, DEPT_ID, PROJ_DESC,
       ACT_START_DATE, ACT_END_DATE, EST_START_DATE, EST_END_DATE
  INTO :PROJ_ID, :EMP_FNAME, :EMP_LNAME, :DEPT_ID, :PROJ_DESC,
       :ACT_START_DATE :ACT-START-DATE-IND,
       :ACT-END-DATE :ACT-END-DATE-IND,
       :EST-START-DATE :EST-START-DATE-IND,
       :EST-END-DATE :EST-END-DATE-IND
  FROM PROJECT, EMPLOYEE
  WHERE PROJ_LEADER_ID = EMP_ID
       AND PROJ_ID = :PROJECT_NUMBER
END-EXEC
```
Retrieving Values through the Command Facility

The following SELECT statement retrieves project information for each employee and consultant in department 1100.

```sql
select e.proj_id, emp_lname, emp_fname, est_start_date, act_start_date
from employee e, project p
where e.proj_id = p.proj_id
  and dept_id = 1100
union select c.proj_id as "Project ID",
    con_lname as "Last Name",
    con_fname as "First Name",
    est_start_date as "Estimated Start Date",
    act_start_date as "Actual Start Date"
from consultant c, project p
where c.proj_id = p.proj_id
  and dept_id = 1100
order by 1, 2, 3;
```

Selecting Multiple Rows into a Buffer

The following SELECT statement returns information on the cost of insurance plans in Massachusetts into an array identified by the host variable :INS-COST-BUFFER:

```sql
EXEC SQL
  SELECT PLAN_CODE, COMP_NAME, MAX_LIFE_COST, FAMILY_COST, DEP_COST
    BULK :INS-COST-BUFFER
  FROM INSURANCE_PLAN
  WHERE STATE = 'MA'
END-EXEC
```

More Information

- For more information about host variables, local variables, or routine parameters, see Host Variables, Local Variables, or Routine Parameters (see page 84).
- For more information about compatible data types for assignment operations, see Comparison, Assignment, Arithmetic, and Concatenation Operations.
- For more information about outer joins, see Expansion of Joined-table, and Query Specifications.
- For more information about the UNION operator, see Expansion of Query-expression.
- For more information about bulk processing in an application program, see the CA IDMS SQL Programming Guide.
SET ACCESS MODULE

The SET ACCESS MODULE statement overrides the default access module to be used by a transaction. You can issue only one SET ACCESS MODULE statement in any given transaction. It is also a CA IDMS extension of the SQL standard. You can use this statement only in SQL that is embedded in a program.

Authorization

None required.

Syntax

```
SET ACCESS MODULE [ access-module-name ] [ :access-module-variable-name ]
```

Parameters

`access-module-name`

Specifies the access module to be used by the current transaction. 
`access-module-name` must identify an access module stored in the dictionary.

`:access-module-variable-name`

Identifies a host variable, local variable, or routine parameter containing the name of the access module to be used by the current transaction. 
`access-module-variable-name` must be a variable previously defined in the application program or SQL routine.

If `access-module-variable-name` is a local variable or routine parameter, the colon must not be coded.
Usage

Order of Execution

If used, the SET ACCESS MODULE statement must be executed before any statement in the transaction other than:
- CONNECT
- SET SESSION
- SET TRANSACTION

Default Access Module

By default, a transaction uses the access module associated with the application program issuing the first SQL statement executed within the SQL session.

One Access Module for a Transaction

A transaction can use only one access module.

Example

Setting the Access Module

The following SET ACCESS MODULE specifies that the current transaction is to use the access module identified by the host variable TRANS-ACC-MOD:

EXEC SQL
  SET ACCESS MODULE :TRANS-ACC-MOD
END-EXEC

Note: For more information about setting the access module for a transaction, see the CA IDMS SQL Programming Guide.

SET host-variable Assignment

The SET host-variable statement enables directly assigning the results of an SQL value expression to a host variable. This statement can only be used in embedded SQL.

Syntax

\[
\text{SET } \text{host-variable} \quad = \quad \text{value-expression} \quad \text{NULL}
\]
Parameters

**host-variable**

Identifies a **host-variable** that is to receive the value of the specified value expression or null. **Host-variable** must be a host variable previously declared in the application program.

**value-expression**

Specifies the value to be assigned to the destination or receiving field of the assignment statement.

**NULL**

Specifies that **host-variable** is set to the NULL value.

Usage

The rules for assignment are provided in Comparison, Assignment, Arithmetic, and Concatenation Operations.

Example

The **host-variable** COMB-NAME is constructed from the values in the **host-variables** FIRST-NAME and LAST-NAME.

```
EXEC SQL
  set:COMB-NAME=trim(:FIRST-NAME) || ' ' || trim(:LAST-NAME);
END-EXEC
```

SET SESSION

The SET SESSION management statement establishes SQL session characteristics. Using the SET SESSION statement, you can perform the following tasks:

- Specify whether subsequent SQL statements must comply with a particular SQL standard
- Change the current schema in effect for the SQL session
- Establish default transaction options
- Control SQL dynamic statement caching
- Specify the encoding of XML values

These session characteristics apply only to SQL submitted through the Command Facility or for dynamic compilation during the execution of an application program.

A SET SESSION statement must include at least one parameter and is a CA IDMS extension of the SQL standard.
Authorization

None required.

Syntax

```
SET SESSION
```

```
session-attribute
```

Expansion of session-attribute

```
CHECK SYNTAX  SQL89
FIPS
CURRENT SCHEMA
schema-name
CURSOR STABILITY
TRANSIENT
READ
READ ONLY
READ WRITE
SQL CACHING
ON
OFF
DEFAULT
XML ENCODING
UTF8
UTF16BE
UTF16LE
EBCDIC
```

Parameters

Parameters for Expansion of session-attribute

**CHECK SYNTAX**

Specifies whether CA IDMS is to check subsequent SQL statements for compliance with a particular standard.

If CHECK SYNTAX is not specified, SQL statements are checked for compliance with CA IDMS Extended SQL.

**SQL89**

Directs CA IDMS to use ANSI X3.135-1989 (Rev), *Database Language SQL with integrity enhancement*, as the standard for compliance.

**FIPS**

Directs CA IDMS to use FIPS PUB 127-1, *Database Language SQL*, as the standard for compliance.

Note: The FIPS standard is based on ANSI X3.135-1989 (Rev). Specifying FIPS in the CHECK SYNTAX parameter has the same effect as specifying SQL89.

**EXTENDED**

Directs CA IDMS to check subsequent SQL statements for compliance with CA IDMS Extended SQL.
CURRENT SCHEMA

Changes the default schema specification for the SQL session.

**schema-name**

Specifies a schema to be used as the default for the SQL session. The specified schema overrides the default in effect for the user session.

**NULL**

Directs CA IDMS to use the default schema in effect for the user session as the default for the SQL session.

CURSOR STABILITY/TRANSIENT READ

Directs CA IDMS to set the default isolation level to that specified.

READ ONLY/READ WRITE

Directs CA IDMS to set the default transaction mode to that specified.

SQL CACHING

Enables you to control dynamic SQL statement caching.

**ON**

If SQL caching is globally enabled, the session will use caching until the session option is changed or until the caching is disabled at the system level.

**OFF**

Regardless of the global setting of SQL caching, the session will not use caching until the session option is changed.

**DEFAULT**

Same as ON.

XML ENCODING

Specifies the type of encoding to use for XML values.

XML ENCODING remains valid until the end of session or until a new SET SESSION command is executed.

**UTF8**

Specifies UTF-8 Unicode encoding.

**UTF16BE**

Specifies UTF-16 Big Endian Unicode encoding.

**UTF16LE**

Specifies UTF-16 Little Endian Unicode encoding.

**EBCDIC**

Specifies EBCDIC encoding. This is the default.
Usage

Default Schema for a User Session

The default schema in effect for a user session is established by a user profile, a system profile, or a DCUF SET PROFILE command.

Duration of SQL Session Characteristics

The SQL session characteristics established by the SET SESSION statement remain in effect until the end of the SQL session or until changed by a subsequent SET SESSION statement.

Precompiled Statements

The SET SESSION command does not cause CA IDMS to check precompiled SQL statements. Use the corresponding precompiler option to enable standards checking for embedded SQL statements.

Establishing Default Transaction Options

You can establish default transaction options for an SQL session using the SET SESSION statement. You can establish the default mode in which a database is accessed (READ ONLY or READ WRITE) and specify an isolation level (CURSOR STABILITY or TRANSIENT READ).

If you do not specify either of these options, the defaults are READ WRITE and CURSOR STABILITY, or the settings specified as part of the access module definition for embedded SQL. The default options may be overridden for an individual transaction by using the SET TRANSACTION statement.

If a transaction is active at the time these options are changed, they impact only subsequent transactions.

Note: For more information about transaction mode and isolation level, see CREATE ACCESS MODULE.
Examples

Checking Compliance with SQL Standard

The following SET SESSION statement which is embedded in an application program, directs CA IDMS to flag any subsequent statements submitted for dynamic compilation that do not comply with SQL standard 89:

```
EXEC SQL
   SET SESSION CHECK SYNTAX SQL89
END-EXEC
```

Setting a Default Schema

The following SET SESSION statement (submitted through the Command Facility) directs CA IDMS to use the SALES_SCH schema as the default schema for the remainder of the SQL session:

```
set session current schema sales_sch;
```

Encoding XML Values

The following examples illustrate EBCDIC and Unicode encoding.

Example 1 - EBCDIC encoding

```
set session XML ENCODING ebc dic;
select cast(SLICE as BIN (27)) as EBCDIC
   from SYSCA.XMLSLICE
   where SLICESIZE = 27 and XMLVALUE =
      XMLCOMMENT(' 0123456789ABCDEF ');
```

The result looks like this:

```
*+ EBCDIC
*+ ---------
*+ 4C5A606040F0F1F2F3F4F5F6F7F8F9C1C2C3C4C5C6404060606E
```
Example 2 - UTF-8 encoding

```sql
set session XML ENCODING UTF8 ;
*+ Status = 0        SQLSTATE = 00000
select cast(SLICE as BIN (27)) as "UTF-8"
  from SYSCA.XMLSLICE
where SLICESIZE = 27 and XMLVALUE =
  XMLCOMMENT(' 0123456789ABCDEF ');
```

The result looks like this:

```plaintext
*+ UTF-8
*+ 3C212D2D2031323334353637383941424344454620202D2D3E
```

Example 3 - UTF-16 Big Endian encoding

```sql
set session XML ENCODING UTF16BE ;
*+ Status = 0        SQLSTATE = 00000
select cast(SLICE as BIN (27)) as "UTF-16 BE"
  from SYSCA.XMLSLICE
where SLICESIZE = 27 and XMLVALUE =
  XMLCOMMENT(' 0123456789ABCDEF ');
```

The result looks like this:

```plaintext
*+ UTF-16 BE
*+ 003C0021002D002D002000200020003000310032003300340035003600
*+ 37003800390041004200430044004500460020002000200020002003E
```

Example 4 - UTF-16 Little Endian encoding

```sql
set session XML ENCODING UTF16LE ;
*+ Status = 0        SQLSTATE = 00000
select cast(SLICE as BIN (27)) as "UTF-16 LE"
  from SYSCA.XMLSLICE
where SLICESIZE = 27 and XMLVALUE =
  XMLCOMMENT(' 0123456789ABCDEF ');
```

The result looks like this:

```plaintext
*+ UTF-16 LE
*+ 3C0021002D002D00200020002000300031003200330034003500360037
*+ 003800390041004200430044004500460020002000200020002003E00
```
More Information

- For more information about CA IDMS compliance with SQL standard SQL, see Summary Comparison to SQL Standard.
- For more information about user profiles, see the CA IDMS Security Administration Guide.
- For more information about system profiles, see the CA IDMS System Tasks and Operator Commands Guide.
- For more information about the DCUF SET PROFILE command, see the CA IDMS System Tasks and Operator Commands Guide.
- For more information about managing SQL sessions, see the CA IDMS SQL Programming Guide.
- For more information about precompiler options, see the CA IDMS SQL Programming Guide.
- For more information about dynamic SQL statement caching, see SQL Cache Tables, and the CA IDMS SQL Programming Guide.

SET TRANSACTION

The SET TRANSACTION management statement overrides the default characteristics of a transaction. The default characteristics are established during access module compilation or, for transactions initiated by the Command Facility, by CA IDMS, and may subsequently have been overridden by a SET SESSION statement.

You can issue only one SET TRANSACTION statement in any given transaction.

Authorization

None required.

Syntax

```sql
SET TRANSACTION

READ ONLY
READ WRITE

CURSOR STABILITY
TRANSIENT READ
```
Parameters

For the duration of the transaction in which the statement is executed, SET TRANSACTION parameters override the defaults. A SET TRANSACTION statement must specify at least one parameter; the combination of READ WRITE and TRANSIENT READ is invalid.

Usage

Order of Execution

If used, the SET TRANSACTION statement must be executed before any statement in the transaction other than:

- CONNECT
- SET ACCESS MODULE
- SET SESSION

Default Transaction Characteristics

Default transaction characteristics are initially established during access module compilation. If not specified as parameters on a CREATE or ALTER ACCESS MODULE statement, and for transactions initiated through the Command Facility, the default transaction characteristics are:

- READ WRITE
- CURSOR STABILITY

These initial defaults can be changed by issuing a SET SESSION statement.

Example

Setting Isolation Level

The following SET TRANSACTION statement specifies that the transaction has an isolation level of transient read:

EXEC SQL
  SET TRANSACTION
    TRANSIENT READ
END-EXEC

More Information

- For more information about isolation levels and SET TRANSACTION parameters, see CREATE ACCESS MODULE.
- For more information about managing transactions, see the CA IDMS SQL Programming Guide.
**SUSPEND SESSION**

The SUSPEND SESSION management statement suspends an SQL session and any transaction currently active within the session. You use the SUSPEND SESSION statement primarily in pseudoconversational programming. It is also a CA IDMS extension of the SQL standard.

**Authorization**

None required.

**Syntax**

```
  SUSPEND SESSION
```

**Usage**

**Effect of SUSPEND SESSION**

When you suspend an SQL session, CA IDMS releases all resources except those required to re-establish the session. Resources required to re-establish the session include locks held by any currently active transaction, cursor currencies, and temporary tables, and dynamically prepared statements.

The SUSPEND SESSION statement does *not* cause a commit or rollback of changes to the database.

**Valid SQL Statement After SUSPEND SESSION**

The first SQL statement you issue after SUSPEND SESSION must be RESUME SESSION.

**Example**

**Suspending a Session**

The following SUSPEND SESSION statement suspends the current SQL session:

```
  EXEC SQL
  SUSPEND SESSION
END-EXEC
```

**More Information**

- For more information about resuming a suspended session, see RESUME SESSION.
- For more information about managing SQL sessions, see the *CA IDMS SQL Programming Guide*. 
TRANSFER OWNERSHIP

The TRANSFER OWNERSHIP authorization statement passes ownership of a schema from one user or group of users to another. It is also a CA IDMS extension of the SQL standard.

Authorization

To issue a TRANSFER OWNERSHIP statement, you must own the schema named in the statement or hold the DBADMIN privilege on the database in which the schema is defined.

Syntax

```
TRANSFER OWNERSHIP OF SCHEMA schema-name
TO authorization-identifier
```

Parameters

**OF SCHEMA schema-name**

Specifies the schema whose ownership is being transferred. *Schema-name* must identify a schema defined in the dictionary.

**TO authorization-identifier**

Identifies the user or group of users to whom you are transferring ownership of the named schema. For expanded *authorization-identifier* syntax, see Expansion of Authorization-identifier.

Usage

**Schema Ownership**

At any given time, a schema can be owned by one user or group of users. The initial owner is the user who created the schema. When ownership of a schema is transferred to a group, each user in the group has all the privileges associated with ownership.

**Ownership of Other Entities**

Technically, schemas are the only database entities that users own. However, by association, the user or group that owns a schema is also said to own the entities in the schema.

**Ownership Privileges**

The owner of a schema has all applicable privileges on entities in the schema, as well as the privilege of granting those privileges to other users or groups. If you transfer ownership of a schema to another user or group, you no longer have any privileges on the entities in the schema.
Examples

Transferring Ownership to a Single User

The following TRANSFER OWNERSHIP statement transfers ownership of the PKE_SCH schema to user PKE:

```
transfer ownership of schema pke_sch
to pke;
```

Transferring Ownership to a Group

The following TRANSFER OWNERSHIP statement transfers ownership of the SALES_SCH schema to the SALES_GRP group:

```
transfer ownership of schema sales
to sales_grp;
```

More Information

- For more information about creating a schema, see CREATE SCHEMA.
- For more information about schema ownership, see your security administrator.

UPDATE

The UPDATE statement is a data manipulation statement that modifies the values in one or more rows of a table.
Authorization

To issue an UPDATE statement, you must:

- Hold the UPDATE privilege on or own the table, view, or table procedure named as the target of the update operation
- Hold the SELECT privilege on or own each table, view, function, procedure and table procedure explicitly named in a subquery in the search condition in the WHERE parameter

Additional authorization requirements apply to:

- A view named in table-reference; each view named in the FROM parameter of such a view; each view named in the FROM parameters of those views, and so forth. For any such view, the owner of the view must hold the grantable UPDATE privilege on or own the table, view, or table procedure named in the FROM parameter of the view definition.
- Each view named in the FROM parameter of a subquery in the search condition; each view named in the FROM parameter of such a view; each view named in the FROM parameters of those views, and so forth. For any such view, the owner of the view must hold the grantable SELECT privilege on or own each table, view, function, or table procedure named in the FROM parameter of the view definition.

Syntax

```
UPDATE table-reference alias

SET column-name = value-expression
    NULL
    (query-expression)

WHERE search-condition
current OF cursor-name
dynamic-name-clause
```

Expansion of dynamic-name-clause

```
LOCAL GLOBAL cursor-name
```
Parameters

**table-reference**
Specifies the table, view, or table procedure whose rows are to be updated. Table-reference must not specify a procedure. If table-reference identifies a view:
- The view must be updateable
- The applicable rows are updated in the table from which the view is derived

For expanded table-reference syntax, see Expansion of Table-reference.

**alias**
Defines a new name to be used to identify the table, view or table procedure within the UPDATE statement. Alias must be a 1- through 18-character name that follows the conventions for SQL identifiers.

**SET**
Specifies the columns to be updated and the value to be stored in each column.

**column-name** =
Identifies a column to be updated. Column-name must identify a column in the table, view, or table procedure named in the UPDATE statement. Column-name must be unique within the SET parameter.

In an UPDATE statement that includes the WHERE CURRENT OF cursor-name parameter, column-name must identify a column specified in the FOR UPDATE parameter of the DECLARE CURSOR statement that defines the named cursor.

**value-expression**
Specifies the value to be stored in the named column. The data type of the value represented by value-expression must be compatible with the data type of the named column. For expanded value-expression syntax, see Expansion of Value-expression.

**NULL**
Directs CA IDMS to store a null value in the named column. The column must be defined to allow null values.
**query-expression**

Represents a value to be used for a column in an UPDATE column statement. The query-expression must return at most, one row and the result table of the query-expression must consist of a single column.

**Note:** For more information about expanded query-expression syntax, see Chapter 8.

**WHERE**

Restricts the rows to be updated. If the UPDATE statement does not include the WHERE parameter, CA IDMS updates all rows in the specified table or view.

**search-condition**

Specifies criteria a row must meet to be updated:

- When the value of search-condition is true, the row is updated
- When the value of search-condition is false, the row is not updated

For expanded search-condition syntax, see Expansion of Search-condition.

**CURRENT OF**

Specifies that only the row that corresponds to the current row of the named cursor is to be updated.
cursor-name

Identifies the cursor whose current row will be updated. **Cursor-name** must identify an open cursor previously defined by a DECLARE CURSOR statement within the application program or by an ALLOCATE CURSOR statement executed within the same SQL transaction.

**Note:** This option may only be used in an UPDATE statement embedded in an application program.

dynamic-name-clause

Identifies the cursor whose current row will be updated.

**Note:** This option may only be used in an UPDATE statement dynamically compiled using a PREPARE or EXECUTE IMMEDIATE statement.

Parameters for Expansion of dynamic-name-clause

**LOCAL**

Indicates that the named cursor has a local scope and was defined using a DECLARE CURSOR statement or an ALLOCATE CURSOR statement. The default is LOCAL.

**GLOBAL**

Indicates that the named cursor was created by an ALLOCATE CURSOR statement and is global in scope.

cursor-name

Specifies the name of the cursor as an identifier. **Cursor-name** must identify an open cursor previously defined by a DECLARE CURSOR statement within the application program or by an ALLOCATE CURSOR statement executed within the same SQL transaction.
Usage

Searched Updates

An UPDATE statement that includes the WHERE search-condition parameter or does not include the WHERE parameter at all is called a searched update. Searched updates may be entered through the Command Facility, executed dynamically, and embedded within application programs.

Positioned Updates

An UPDATE statement that includes the WHERE CURRENT OF cursor-name parameter is called a positioned update. Positioned updates are valid only from within an application program.

Dynamic Positioned Updates

A dynamic positioned UPDATE statement is one that references a dynamic cursor. Such an UPDATE statement may be embedded within an application program or created dynamically using a PREPARE or EXECUTE IMMEDIATE statement.
A positioned UPDATE statement embedded in an application program may reference a static cursor or a dynamic cursor. A positioned UPDATE statement created dynamically using a PREPARE or EXECUTE IMMEDIATE statement can only reference a dynamic cursor.

**Ambiguous Cursor References**

When a dynamic positioned UPDATE statement is being created by a PREPARE or EXECUTE IMMEDIATE statement, it is possible that CA IDMS may not be able to determine which cursor is being referenced. This will occur if the application program contains a DECLARE CURSOR statement that defines a cursor having the referenced name and the program has also executed an ALLOCATE cursor statement that creates a cursor with the same name and a local scope. Under these conditions, CA IDMS cannot determine which of the two cursors is being referenced. To avoid such problems, it is advisable to use different names for cursors that are declared from those that are allocated with a local scope.

**Restrictions on Table-reference**

In a searched update, the table, view, or table procedure named in the UPDATE statement cannot also be named in the FROM parameter of any subquery included in the specified search condition; or, in the case of a view, in any search condition used in the view definition. The same restriction applies for any update that uses a subquery as the value to be stored in an updated column. Therefore, you cannot update data in a table from which you select in a subquery.

In a positioned update, the table, view, or table procedure named in the UPDATE statement must also be named in the FROM parameter of the query specification used in the definition of the named cursor.
Restriction on Value-expression

The value expression that specifies the value to be stored in a column cannot include any aggregate functions.

Cursor Position after a Positioned Update

After a positioned update, the position of the cursor named in the UPDATE statement remains unchanged.

Restrictions for Tables in Referential Constraints

If the table named in an UPDATE statement is the referencing table in a referential constraint, CA IDMS will update a row in the table only if, after the update operation, the foreign-key columns in the row satisfy either of the following conditions:

- The columns must be all or partially null
- The foreign-key values must match the referenced-column values in a row of the referenced table

If the table named in an UPDATE statement is the referenced table in a referential constraint, and the referencing table includes one or more rows whose foreign-key values match the referenced-column values of the row in the referenced table to be updated, CA IDMS will update the row only if the update operation does not change the values in the referenced columns.

Satisfying Check Constraints

If the updates to a row do not satisfy the check constraints, if any, in the table definition, CA IDMS returns an error and does not update the row.

Updating Through a View

If the target of the update statement is a view, the view must be updateable, and only rows that can be retrieved through the view can be updated through the view.

If the view being updated is defined with WITH CHECK OPTION, any WHERE clause in the view definition, or in the definitions of any other views nested within its definition, will be applied like a check constraint to restrict the update values.

Using a query-expression as a Source Value

If a query-expression used as the value stored in a column returns no rows, the column is set to the null value. If the column does not allow nulls, an exception is raised.
Examples

Requesting a Searched Update

The following UPDATE statement updates the MANAGER_ID column in the EMPLOYEE table for rows where the value in the column currently is 3222:

```sql
update employee
    set manager_id = 9847
    where manager_id = 3222;
```

Requesting a Positioned Update

The following UPDATE statement updates the BENEFITS table through the BONUS_CURSOR cursor. The statement stores the value in the host variable CALC-BONUS-AMT in the BONUS_AMOUNT column of the table row that corresponds to the current row of the cursor.

```sql
EXEC SQL
    UPDATE BENEFITS
        SET BONUS_AMOUNT = :CALC-BONUS-AMT
    WHERE CURRENT OF BONUS_CURSOR
END-EXEC
```

A Positioned UPDATE Referencing a DECLAREd Cursor

The following statement updates the current row of the cursor C1. C1 may be a dynamic or static cursor, but it must have been defined using a DECLARE CURSOR statement. Furthermore, the cursor-specification on which C1 is based must contain a FOR UPDATE option which directly or implicitly includes the EMP_LNAME column:

```sql
EXEC SQL
    UPDATE EMPLOYEE
        SET EMP_LNAME = :emp-name
    WHERE CURRENT OF C1
END-EXEC
```

A Positioned UPDATE Referencing an ALLOCATEd Cursor

The following statement updates the current row of a cursor whose name is specified in the variable CNAME. The referenced cursor must have been defined using an ALLOCATE CURSOR statement:

```sql
EXEC SQL
    UPDATE EMPLOYEE
        SET EMP_LNAME = :emp-name
    WHERE CURRENT OF GLOBAL :CNAME
END-EXEC
```
A Dynamically-compiled Positioned UPDATE Statement

The following statement updates the current row of local cursor C1. C1 may have been defined using either a DECLARE CURSOR statement or an ALLOCATE CURSOR statement. In either case, the cursor name in the UPDATE statement is specified as an identifier rather than as a literal or host variable:

```
EXEC SQL
  EXECUTE IMMEDIATE
    'UPDATE EMPLOYEE SET EMP_STATUS = "T"
      WHERE CURRENT OF LOCAL C1'
END-EXEC
```

**Note:** The keyword LOCAL is unnecessary since it is the default. Regardless of whether it is specified, if two local cursors named C1 have been defined, one using a DECLARE CURSOR statement and one using an ALLOCATE CURSOR statement, the EXECUTE IMMEDIATE statement will fail on an ambiguous cursor error.

Using query-expressions to Update Columns

The following example sets the value of the SALARY_BUDGET column in the DEPARTMENT table based on the current salaries of all employees in the department.

```
update department d
  set salary_budget =
    (select 1.1 * sum (salary) from employee e
      where e.deptid = d.deptid)
```

Updating All Rows

The following UPDATE statement modifies every row in the INSURANCE_PLAN table. The statement increases all the values in the FAMILY_COST column by 2 percent and all the values in the DEP_COST column by 1 percent:

```
update insurance_plan
  set family_cost = family_cost * 1.02,
      dep_cost = dep_cost * 1.01;
```
WHENEVER

More Information

- For more information about updateable views, see CREATE VIEW.
- For more information about updateable result tables, see DECLARE CURSOR.
- For more information about compatible data types for assignment operations, see Comparison, Assignment, Arithmetic, and Concatenation Operations.
- For more information about null values, see Null Values.
- For more information about defining and manipulating cursors, see CLOSE, DECLARE CURSOR, FETCH, and OPEN (see page 494).
- For more information about referential constraints, see CREATE CONSTRAINT.

WHENEVER

The WHENEVER precompiler-directive statement specifies an action to be taken when the execution of an SQL statement results in a nonzero SQLCODE value. The WHENEVER statement directs the precompiler to insert the appropriate conditional code after each subsequent SQL statement that generates a call to CA IDMS.

You can use this statement only in SQL that is embedded in a program.

Authorization

None required.

Syntax

```
WHENEVER [ NOT FOUND ] [ SQLERROR | SQLWARNING ] [ CONTINUE | GO TO ] [ label ] [ CALL subroutine-name ]
```
**Parameters**

**NOT FOUND**
Directs CA IDMS to take the specified action when the execution of an SQL statement results in an SQLCODE value of 100.

**SQLERROR**
Directs CA IDMS to take the specified action when the execution of an SQL statement results in an SQLCODE value that is less than zero.

**SQLWARNING**
Directs CA IDMS to take the specified action when the execution of an SQL statement results in an SQLCODE value of 1.

SQLWARNING is a CA IDMS extension of the SQL standard.

**CONTINUE**
Specifies that processing is to continue with the next statement.

**GO TO label/label**
Specifies that processing is to continue with the first statement at the named label. Label must be the name of a section or the unqualified name of a paragraph in the application program.

GO TO and GOTO are synonyms and can be used interchangeably. Label and :label are synonyms and can be used interchangeably.

The GO TO parameter is not valid in CA ADS application programs.

The specification of a label without a colon (:) is a CA IDMS extension of the SQL standard.

**CALL subroutine-name**
Specifies that processing control is to pass to the named subroutine. Subroutine-name must identify a subroutine subsequently defined in the process module.

The CALL parameter is valid only in CA ADS process modules.

The CALL parameter is a CA IDMS extension of the SQL standard.

**Usage**

**Scope of the WHENEVER Statement**

The WHENEVER statement for a specified condition applies to all subsequent SQL statements until the precompiler encounters another WHENEVER statement that names the same condition.
Example

Specifying Error Processing

The following WHENEVER statement specifies that control is to pass to the section or paragraph named SQLCODE-CHECK whenever CA IDMS returns a negative value in SQLCODE:

EXEC SQL
    WHENEVER SQLERROR
        GO TO :SQLCODE-CHECK
END-EXEC

More Information

- For more information about SQLCODE values, see SQLCODE Values.
- For more information about COBOL section and paragraph names, refer to the appropriate COBOL documentation.
This section contains the following topics:

- **Overview** (see page 559)
- **SQL Control Statements** (see page 560)
- **CALL** (see page 561)
- **CASE** (see page 562)
- **Compound Statement** (see page 566)
- **EXEC ADS** (see page 573)
- **IF** (see page 577)
- **ITERATE** (see page 580)
- **LEAVE** (see page 583)
- **LOOP** (see page 585)
- **REPEAT** (see page 587)
- **RESIGNAL** (see page 589)
- **RETURN** (see page 591)
- **SET Assignment** (see page 592)
- **SIGNAL** (see page 594)
- **WHILE** (see page 597)

## Overview

The statements defined in the Control category are the basis for the SQL procedural language used by SQL routines. An SQL routine usually contains procedural language statements and data manipulation statements. It can also use any statement as specified by the `procedure-statement` syntax.

**Note:** For more information, see CREATE PROCEDURE or CREATE FUNCTION.
SQL Control Statements

The statements of the Control category include syntax to perform the following:

- Direct the flow of control
- Assign the result of expressions to variables and parameters
- Specify condition handlers to process various conditions
- Signal and resignal conditions
- Declare local cursors

The advantages for writing SQL routines in the SQL language include:

- Easy readable and simple but powerful programs
- Single Language to access and process data
- Native support for all the SQL data types makes handling of VARCHAR, DATE, TIME, and TIMESTAMP data a lot easier
- Built-in NULL support avoids the burden of having to define and manipulate NULL indicators for nullable data such as table columns or parameters of SQL routines which are always nullable
- Flexible handlers are able to process SQL events easily
- A single development and test platform fully integrated in all CA IDMS supported environments

### SQL Control Statements

All SQL control statements are programmatic only. The following table provides a brief description of the SQL control statements.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>Invokes an SQL procedure.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> The CALL statement is also a DML statement. The syntax and semantics of the CALL control statement are a subset of the CALL DML statement.</td>
</tr>
<tr>
<td>CASE</td>
<td>Determines the execution flow by the evaluation of one or more value-expressions.</td>
</tr>
<tr>
<td>Compound</td>
<td>Specifies a grouping of statements, with optional definitions of local variables, cursors, and handlers.</td>
</tr>
<tr>
<td>EXEC ADS</td>
<td>Starts a block of CA ADS code.</td>
</tr>
<tr>
<td>IF</td>
<td>Determines by evaluation of a search-condition, which block of statements are executed.</td>
</tr>
<tr>
<td>ITERATE</td>
<td>Begins a new iteration in a programmatic loop.</td>
</tr>
</tbody>
</table>
## CALL

The CALL control statement invokes SQL-invoked procedures. The syntax and semantics are a subset of the CALL DML statement.

### Authorization

See Authorization.

### Syntax

```
CALL procedure-reference
```

### Parameter

**procedure-reference**

Identifies the procedure that is invoked, the input values that pass to the procedure and optionally the local variables and routine parameters for passing and returning values of input/output parameters.

### Usage

See Usage.
Example

The function GET_NAME invokes the SQL procedure GET_FIRST_LAST to retrieve the first and last names of the employee’s empid that is specified as the input parameter. The function then returns the combined and trimmed first and last names as one string.

```sql
set options command delimiter '++';
create procedure GET_FIRST_LAST
    ( P_EMPID NUMERIC(4)
    , P_FNAME char(20)
    , P_LNAME char(30))
    EXTERNAL NAME DEMOGTFL LANGUAGE SQL
/*
** Get first and last name of employee
*/
Select EMP_FNAME, EMP_LNAME into P_FNAME, P_LNAME From DEMOEMPL.EMPLOYEE
    where EMP_ID = P_EMPID
++
commit++

create function GET_NAME
    (P_ID NUMERIC(4))  RETURNS varchar(40)
    EXTERNAL NAME DEMOGETN LANGUAGE SQL
begin not atomic
/*
** Get name of employee
*/
declare FNAME char(20);
declare LNAME char(20);
call GET_FIRST_LAST(P_ID, FNAME, LNAME);
return trim(FNAME)|| ' ' || trim(LNAME);
end++
set options command delimiter default++
commit;

select GET_NAME(5008) from SYSCA.SINGL
set options command delimiter default++
commit;
```

```
select GET_NAME(5008) from SYSCA.SINGL_NULL;
*+
*+ USER_FUNC
*+ ---------------------
*+ Timothy Fordman
*+
*+ 1 row processed
```

CASE

The CASE statement selects different execution paths depending on the evaluation of one or more value-expressions.
## Syntax

```plaintext
case 
<table>
<thead>
<tr>
<th>simple-case-when-clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>searched-case-when-clause</td>
</tr>
<tr>
<td>end case</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>procedure-statement</td>
</tr>
<tr>
<td>;</td>
</tr>
</tbody>
</table>
```

Expansion of simple-case-when-clause

```plaintext
value-expression
```

Expansion of searched-case-when-clause

```plaintext
value-expression
```

## Parameters

### Parameters for Expansion of simple-case-when-clause

**CASE value-expression**

Specifies the value expression whose outcome is compared to the outcomes of the **value-expressions** in the **WHEN** clauses.

**WHEN value-expression**

Specifies a value expression whose outcome is compared to the outcome of the **CASE value-expression**. If the two values are equal, the group of statements specified in the corresponding **THEN** is executed.

**THEN procedure-statement**

Identifies the group of statements to be executed when the value expressions of the **CASE** and **WHEN** clauses are equal.

### Parameters for Expansion of searched-case-when-clause

**CASE WHEN**

Identifies the **CASE** as a searched case.
WHEN search-condition

Specifies the search condition whose outcome, if true, results in the execution of the group of statements specified by the THEN clause.

THEN procedure-statement

Identifies the group of statements executed when the search-condition in the corresponding WHEN clause evaluates to true.

ELSE procedure-statement END CASE

Specifies the group of statements to be executed when none of the THEN group of statements has been executed because of the evaluation and comparison of the value-expression's and search-condition's. This clause can be specified for both simple and searched case statements.

Usage

SQL Exceptions

If an ELSE clause is not specified and none of the THEN group of statements has been executed because of the outcome of evaluation of the value expressions and search conditions, an SQL exception is raised.
Examples

The first example demonstrates the use of a simple-case-when-clause.

```
set options command delimiter '++';
create function USER01.TCASE1
   ( TITLE     varchar(40) with default
     , P_EMP_ID  unsigned numeric(4)
   ) RETURNS   varchar(30)
     external name TCASE1 language SQL
begin not atomic
   /*
   ** Function selects an employee with the given EMP_ID and swaps
   ** the first_name value 'James' with 'Jim'.
   ** Returns a message text with the outcome of the execution
   */
   declare MY_STATUS varchar(30);
   declare LOC_FNAME char(20) default ' ';
   select EMP_FNAME into LOC_FNAME
      from DEMOEMPL.EMPLOYEE
      where EMP_ID = P_EMP_ID;
   case LOC_FNAME
      when 'James'
         then update DEMOEMPL.EMPLOYEE set EMP_FNAME = 'Jim'
            where EMP_ID = P_EMP_ID;
            set MY_STATUS = 'James->JIM';
      when 'Jim'
         then update DEMOEMPL.EMPLOYEE set EMP_FNAME = 'James'
            where EMP_ID = P_EMP_ID;
            set MY_STATUS = 'Jim->James';
      when 'Thomas'
         then update DEMOEMPL.EMPLOYEE set EMP_FNAME = 'Thomas'
            where EMP_ID = P_EMP_ID;
            set MY_STATUS = 'Dummy update';
      else set MY_STATUS = 'No Changes';
   end case;
   return MY_STATUS;
end
++
set options command delimiter default++
commit;
```

select USER01.TCASE1('TCASE1', 1034)from SYSCA.SINGLETON_NULL;
++
++ USER_FUNC
++ ---------
++ Jim->James
The second example demonstrates the `searched-case-when-clause`. It is functionally equivalent with the example of `simple-case-when-clause`.

```
set options command delimiter '++';
create function USER01.TCASESR1
  ( TITLE     varchar(40) with default
    , P_EMP_ID  unsigned numeric(4)
  ) RETURNS   varchar(30)
    external name TCASESR1 language SQL
begin not atomic
  /*
   ** Function selects an employee with the given EMP_ID and
   ** does some conditional updates.
   ** Returns a message text with the outcome of the execution
   */
  declare MY_STATUS varchar(30);
  declare LOC_FNAME char(20) default ' ';
  declare LOC_LNAME char(20) default ' ';

  select EMP_FNAME, EMP_LNAME into LOC_FNAME, LOC_LNAME
    from DEMOEMPL.EMPLOYEE where EMP_ID = P_EMP_ID;

  case
    when LOC_FNAME = 'James'
      then update DEMOEMPL.EMPLOYEE set EMP_FNAME = 'Jim'
        Where EMP_ID = P_EMP_ID;
        set MY_STATUS = 'James->JIM';
    when LOC_FNAME = 'Jim' and LOC_LNAME = 'Gallway'
      then update DEMOEMPL.EMPLOYEE set EMP_FNAME = 'James'
        Where EMP_ID = P_EMP_ID;
        set MY_STATUS = 'Jim->James';
    when LOC_LNAME = 'Van der Bilck'
      then update DEMOEMPL.EMPLOYEE set EMP_LNAME = 'Vanderbilck'
        Where EMP_ID = P_EMP_ID;
        set MY_STATUS = 'Van der Bilck->Vanderbilck';
    else set MY_STATUS = 'No Changes';
  end case;

  return MY_STATUS;
end
++
set options command delimiter default++
```
Syntax

```
BEGIN
  ATOMIC
  NOT ATOMIC

  beg-label:

  variable-declaration
    condition-declaration

  cursor-declaration

  handler-declaration

  procedure-statement

END

end-label
```

Expansion of variable-declaration

```
DECLARE -
  variable
  data-type
    DEFAULT
    NULL
    const
```

Expansion of condition-declaration

```
DECLARE -
  condition-name
    CONDITION FOR SQLSTATE
      VALUE
      const
```

Expansion of handler-declaration

```
DECLARE -
  CONTINUE
  HANDLER FOR
    SQLSTATE
    SQLWARNING
    SQLERROR
    NOT FOUND
    SQLSTATE value
      'sqlstate'
    condition-name

procedure-statement
```
Parameters

`beg-label:`

Specifies a 1- through 32-character SQL identifier that labels the compound statement. The value must be different from any other label used in the compound statement.

**ATOMIC**

Specifies that an unhandled exception raised while executing the compound statement causes a rollback of the effects of the compound statement.

**NOT ATOMIC**

Specifies that an unhandled exception raised while executing the compound statement does not cause a rollback of the effects of the compound statement. This is the default.

**variable-declaration**

Defines a local variable.

**condition-declaration**

Defines a name for a condition for the purposes of referencing it in other statements.

**cursor-declaration**

Defines a local cursor for use within the compound statement. For a description of this clause, see DECLARE CURSOR.

**handler-declaration**

Defines a handler routine for SQL exception or completion conditions. A handler routine receives control when the execution of an SQL statement fails or terminates with a condition for which the handler has been defined. The three types of handlers (CONTINUE, EXIT, UNDO) and the conditions under which they are invoked are described in Parameters for Expansion of handler-declaration in this section.

**procedure-statement**

Defines the SQL procedure statement that is to be executed when the handler routine is invoked. Procedure-statement may be any statement except a compound statement.

**end-label**

Specifies an SQL identifier that labels the end of the compound statement. If specified, a `beg-label` must also have been specified and both labels must be equal.
Parameters for Expansion of variable-declaration

**variable**

Specifies the name of the local variable. *Variable* must be a 1- through 32-character name that follows the conventions for SQL identifiers. The names of all local variables declared within a compound statement must be unique.

**data-type**

Specifies a set of values that share processing characteristics. See Expansion of Data-type.

**DEFAULT**

Specifies the initial value of the local variable.

**NULL**

Initializes the local variable to NULL.

**const**

Initializes the local variable to the value of *const*. *Const* must be a literal whose value is compatible for assignment to the local variable.

*Note:* If DEFAULT is not specified, the local variable is not initialized.

Parameters for Expansion of condition-declaration

**DECLARE condition-name FOR CONDITION SQLSTATE**

Defines a name for a condition. This name can be used in other statements to refer to the condition.

**condition-name**

Specifies the name to be assigned to the condition. *Condition-name* must be a 1- through 32-character name that follows the conventions for SQL identifiers. The names of all conditions declared within a compound statement must be unique.

**VALUE**

Specifies an optional keyword without semantic meaning.

**const**

Specifies the value of SQLSTATE that constitutes the condition. *const* is a 5-character string-literal that consists of only digits (0-9) and capital alphabetic characters (A-Z). *const* cannot be '00000', the value of SQLSTATE for successful completion.
Parameters for Expansion of handler-declaration

CONTINUE

After executing the handler action, a CONTINUE handler returns control to the statement following the one that caused the event. If this statement is contained in an IF, CASE, LOOP, WHILE, or REPEAT statement, control is returned to the statement following the IF, CASE, LOOP, WHILE, or REPEAT statement.

EXIT

After executing the handler action, an EXIT handler returns control to the statement following the compound statement. If there is no statement following the compound statement, control is returned to the invoker of the routine.

UNDO

Before executing the handler action, an UNDO handler will rollback the database changes caused by the execution of the compound statement that caused the handler to be activated. After the handler actions have been executed, control is returned to the statement following the compound statement. If there is no statement after the compound statement, control is returned to the invoker of the routine. An UNDO handler requires its defining compound statement to be ATOMIC.

SQLEXCEPTION

Specifies that the handler is to be activated for all events except those of classes "Successful completion" (SQLSTATE = '00xxx'), "Completed with Warning" (SQLSTATE = '01xxx'), and "Completed with No Data" (SQLSTATE = '02xxx').

SQLWARNING

Specifies that the handler is to be activated for events of the class, "Completed with Warning" (SQLSTATE = '01xxx').

NOT FOUND

Specifies that the handler is to be activated for events of the class, "Completed with No Data" (SQLSTATE = '02xxx').
'sqlstate'  

Specifies a value of SQLSTATE for which the handler is activated. 'sqlstate' must be a 5-character string-literal that consists of only digits (0-9) and capital alphabetic characters (A-Z). 'sqlstate' cannot be '00000', the value of SQLSTATE for successful completion.

condition-name  

Specifies the name of a condition for which the handler is activated.  
Condition-name must identify a condition declared in the compound statement.

procedure-statement  

Defines an SQL procedure statement to be included in the compound statement. Procedure-statement may be any statement including a compound statement. For more information, see the expansion for this syntax in CREATE FUNCTION (see page 341) or CREATE PROCEDURE (see page 361).

Usage  

Variables, Parameters, and Column Names  

When ambiguity exists in referencing local variables, parameters and column names, qualification is required to resolve the ambiguity.  

Note: For more information, see Expansion of Local-variable and Expansion of Routine-parameter (see page 85).

Nesting of Compound Statement  

A compound statement cannot contain other compound statements with the exception of handlers. A handler, which necessarily is contained in a compound statement, can have a compound statement as its procedure statement procedure-statement.

Handlers  

When both a generic class handler (a handler for SQLEXCEPTION or SQLWARNING) and a specific handler cover the same event, the more specific handler is invoked when the event occurs.  

Only one handler for a specific event can be defined.  

Handlers cannot be defined with duplicate conditions.  

If an SQL exception occurs in a compound statement for which there is no handler defined, control returns to the statement following the compound statement that caused the exception and an implicit RESIGNAL is executed. The exception is passed in the SQLSTATE. Database changes made by compound statements defined as ATOMIC will be rolled back before control returns.
Atomic Compound Statements

Compound statements defined as ATOMIC cannot contain the transaction management statements, COMMIT and ROLLBACK, or the session management statement, RELEASE.

Cursor state upon exiting from a compound statement

When execution of a compound statement ends, all cursors defined within the compound statement that are still open are automatically closed, except for returnable cursors.

Note: For more information about returnable cursors, see DECLARE CURSOR.

Example

The procedure USER01.TCOMP01 retrieves an employee for a given EMP_ID and returns a formatted name. An exit handler for NOT FOUND handles the NOT FOUND condition. An exit handler for SQLEXCEPTION handles generic database errors.

```sql
set options command delimiter '++';
create procedure USER01.TCOMP01
( P_ID     numeric(4)
, P_NAME   char(30)
, RESULT   varchar(30)
) external name TCOMP01 language SQL
Label_400:
/*
 ** Return formatted name of employee with given EMP_ID
 */
begin not atomic
 declare L_FNAME   char(50);
 declare L_LNAME   char(50);

 declare exit handler for SQLEXCEPTION
 label_8888:
 begin not atomic
 set RESULT = 'Unexpected SQLSTATE: ' || SQLSTATE;
 set P_NAME = '** Error **';
 end;

 declare exit handler for NOT FOUND
 set RESULT = 'No employee for EMP_ID: '
 || cast(P_ID as char(4));
```
set RESULT = ' '; set P_NAME = ' '; select EMP_FNAME, EMP_LNAME into L_FNAME, L_LNAME from DEMOEMPL.EMPLOYEE where EMP_ID = P_ID;
set P_NAME = trim(L_FNAME) || ' ' || trim(L_LNAME);
set RESULT = 'All OK';
End label_400
++
set options command delimiter default++
commit;
call user01.TCOMP01(1003);
*+
*+ P_ID  P_NAME                  RESULT
*+ ----- -----                  -----       
*+ 1003 Jim Baldwin             ALL OK

call user01.TCOMP01(9);
*+
*+ P_ID  P_NAME                  RESULT
*+ ----- -----                  -----       
*+ 9      NO EMPLOYEE FOR EMP_ID: 9

call user01.TCOMP01(-2000);
*+
*+ P_ID  P_NAME                  RESULT
*+ ----- -----                  -----       
*+ -2000 ** ERROR **             UNEXPECTED SQLSTATE: 22005

EXEC ADS

The EXEC ADS statement is a CA IDMS extension that enables inserting CA ADS code in SQL routines.

Syntax

```
EXEC ADS吸附-process-stmnt ;
```

Parameters

**ads-process-stmnt**

Specifies a CA ADS statement to be executed.
Usage

Allowable CA ADS statements

Only CA ADS statements that are allowed in a mapless dialog can be included in the body of an SQL routine.

Care should be taken in coding SQL transaction and session management statements because a ROLLBACK or COMMIT breaks the atomicity of a compound statement containing the EXEC ADS statement.

Referencing SQL-defined data

SQL-defined data can be referenced by respecting the mapping rules for identifiers and data types between SQL and CA ADS:

- Underscore characters are mapped to dashes.
- VARCHAR data are structures that start with a smallint field that holds the length of the character data, followed by the character data itself. The name of the structure is the mapped SQL identifier. The name of the length field is the mapped SQL identifier suffixed with "-LEN". The name of the data field is the mapped SQL identifier suffixed with "-TEXT".
- Nullable SQL data must have their NULL indicators managed properly. All SQL parameters and local variables are nullable.
- Date, time, and timestamp data types must be correctly processed.
Using IDD records and record elements from the dictionary

It is possible to use records and record elements that are defined as IDD records. This requires the specification of "ADD RECORD record name" in the ADS Compile Option of the CREATE PROCEDURE or CREATE FUNCTION statements. See the following example. For more information, see CREATE PROCEDURE and CREATE FUNCTION.

Using EXEC ADS for debugging

Some of the ADS utility commands can be used for debugging SQL routines. SNAP, TRACE, and WRITE TO LOG are of particular interest. See the second example below. All the SQL local variables and internal variables are contained in predefined ADS records. The name of these predefined records is constructed as follows: SQLLOCnnnnxxxxxx with xxxxxxxx representing the external name of the SQL routine and nnnn a four digit number with values starting from 0 for the internal variables to the total count of compound statements.

Assume an SQL routine with an associated external name of 'GETLNAME' containing two compound statements, then the content of all internal and SQL local variables can be dumped to the log as follows:

exec ADS snap record(SQLLOC0000GETLNAME
                      , SQLLOC0001GETLNAME
                      , SQLLOC0002GETLNAME).;

A complete report of the SQL routine, including the layout of all the records can be obtained by executing the batch utility ADSORPTS. For the SQL routine with external name GETLNAME, the control statement input for ADSORPTS would look like the following:

DIALOG=(GETLNAME),REPORTS=ALL

For the TRACE to be functional, the ADS dialog associated with the SQL routine needs to be compiled with symbol table information. This option can be turned on by specifying SYMBOL TABLE IS YES in the ADS Compile Option of the CREATE PROCEDURE or CREATE FUNCTION statements. See the next example. For more information, see CREATE PROCEDURE and CREATE FUNCTION (see page 341).
Example

Using EXEC ADS to obtain the current LTERM

The SQL function USER01.TEXECADS2 returns the LTERM ID of the LTERM on which the function is being executed.

```sql
set options command delimiter '++';
create function USER01.TEXECADS2
  ( P_DUMMY   char(1)
  ) returns   char(8)
  external name TEXECAD2 language SQL
begin not atomic
  /*
  ** SQL Function to return LTERM ID using EXEC ADS
  */
  declare L_LTERMID char (8) default ' ';
  exec ads
    ACCEPT LTERM ID INTO L_LTERMID. ;
  return L_LTERMID;
end
++
set options command delimiter default++
commit;

select USER01.TEXECADS2()
  from SYSCA.SINGLETON_NULL;
++
++ USER_FUNC
++ ........
++ VL71001
```

Using EXEC ADS to debug a SQL routine

In the following example, the ADS COMPILER OPTIONS is used to add the ADSO-APPLICATION-GLOBAL-RECORD so that it can be accessed within the SQL function. Furthermore, the ADS dialog associated with the SQL function is compiled with diagnostics and symbol table so that debugging and diagnostic information is available at run time.

An EXEC ADS statement is placed as the very first executable statement of the SQL function to snap the local variables to the IDMS log to verify the initialization and to turn on ADS tracing. The EXEC ADS statement at the end, snaps the local variables and the ADSO-APPLICATION-GLOBAL-RECORD before returning to the invoker of the function. This statement also turns the ADS tracing off.
set options command delimiter '++';
create function GET_NAME
  (P_ID NUMERIC(4))  RETURNS varchar(40)
    EXTERNAL NAME DEMOGETN LANGUAGE SQL
    ADS COMPIL Options
      symbol table is yes
      diagnostic is yes
      add record ADSO-APPLICATION-GLOBAL-RECORD;
begin not atomic
  /*
    ** Get name of employee
    */
    declare FNAME char(20) default ' ';
    declare LNAME char(20) default ' ';
    declare L_STATEMENT char(160);
    EXEC ADS snap record(SQLLOC0001DEMOGETN).
    trace all.;
    set L_STATEMENT =
    'select EMP_FNAME, EMP_LNAME' ||
    ' from DEMOEMPL.EMPLOYEE where EMP_ID = ?';
    prepare 'DYN1' from L_STATEMENT
    describe output using descriptor SQLDA;
    allocate 'CUR1' cursor for 'DYN1';
    open 'CUR1' using P_ID;
    fetch 'CUR1' into FNAME, LNAME;
    EXEC ADS snap record(SQLLOC0001DEMOGETN).
    snap record(ADSO-APPLICATION-GLOBAL-RECORD).
    trace off.;
    return trim(FNAME)|| ' ' || trim(LNAME);
  end++
set options command delimiter default++
commit;
select GET_NAME(1003) from SYSCA.SINGLETON_NULL;

IF

The IF statement selects different execution paths depending on the evaluation of one or more truth value expressions, given as SQL search conditions.
**Syntax**

```plaintext
IF search-condition THEN procedure-statement ;

ELSEIF search-condition THEN procedure-statement ;

ELSE procedure-statement ; END IF
```

**Parameters**

**IF search-condition**

Specifies the truth value expression to be evaluated. The outcome of the evaluation determines the execution path.

**THEN procedure-statement**

Specifies the statements to be executed if the immediately preceding search condition is true.

**ELSEIF search-condition**

Specifies the truth value expression to be evaluated if the outcomes of all previously evaluated search conditions are false.

**ELSE procedure-statement**

Specifies the statements to be executed if all search conditions are false.

**Usage**

If no alternative execution path is given, execution continues with the next statement outside the IF.
Example

set options command delimiter '++';
create procedure USER01.TIF1
  ( TITLE   varchar(10) with default
    , P_LEFT    integer
    , P_RIGHT   real
    , RESULT  varchar(30)
  )
  EXTERNAL NAME TIF1 LANGUAGE SQL
Label_200:
begin not atomic
  /*
  ** Compare an integer value with a real value
  */
  if (P_LEFT > P_RIGHT)
    then set RESULT = 'p_left > p_right';
  elseif (P_LEFT = P_RIGHT)
    then set RESULT = 'p_left = p_right';
  elseif  (P_LEFT < P_RIGHT)
    then set RESULT = 'p_left < p_right';
  else   set RESULT = 'p_left and/or p_right NULL !';
  end if;
end
++
commit++
call user01.TIF1('Test IF >', 4, 2)++
+
  TITLE    P_LEFT       P_RIGHT    RESULT
  -----    ------   -------    ------
  Test IF >   4   2.0000000E+00  P_LEFT > P_RIGHT

call user01.TIF1('Test IF <', 4, 9)++
+
  TITLE    P_LEFT       P_RIGHT    RESULT
  -----    ------   -------    ------
  Test IF <   4   9.0000000E+00  P_LEFT < P_RIGHT

call user01.TIF1('Test IF =', 2, 2)++
+
  TITLE    P_LEFT       P_RIGHT    RESULT
  -----    ------   -------    ------
  Test IF =   2   2.0000000E+00  P_LEFT = P_RIGHT

call user01.TIF1('Test IF ', 4)++
+
  TITLE    P_LEFT       P_RIGHT    RESULT
  -----    ------   -------    ------
  Test IF 4  <null>  P_LEFT AND/OR P_RIGHT NULL !
set options command delimiter default++
The ITERATE statement terminates execution of the current iteration of an iterated statement, such as LOOP, REPEAT or WHILE. If the iteration condition is true, a new iteration starts; otherwise, the statement following the iterated statement is executed.

**Syntax**

```
 ITERATE stmt-label
```

**Parameters**

`stmt-label`

Specifies the begin label of the iterated statement.

**Usage**

**Statements that may be iterated**

The labeled statement referred in the ITERATE must be a LOOP, REPEAT, or WHILE statement that contains the ITERATE statement.
Example

The procedure USER01.TITERATE1 retrieves all rows of the DEMOEMPL.EMPLOYEE table three times. The first loop uses a WHILE, the second uses a REPEAT, and the third uses a LOOP statement.

```sql
set options command delimiter '++';
create procedure USER01.TITERATE1
  ( TITLE     varchar(10) with default
    , P_FNAME   char(20)
    , P_COUNT     integer
    , RESULT    varchar(10)
  )
  EXTERNAL NAME TITERATE LANGUAGE SQL
Label_600:
  begin not atomic
    declare FNAME   char(20);
    declare LNAME   varchar(20);
    declare EMP1 CURSOR FOR
      Select EMP_FNAME, EMP_LNAME
        From DEMOEMPL.EMPLOYEE;
    /*
     * ITERATE in WHILE
     *
     */
    set RESULT = '?????';
    open EMP1;
    while_loop:
      while (9 = 9)
        do
          fetch EMP1 into FNAME, LNAME;
          if (SQLSTATE = '00000')
            then
              set P_COUNT = P_COUNT + 1;
              iterate while_loop;
            end if;

            if (SQLSTATE = 'abcde')
              then
                iterate while_loop;
              end if;

            set RESULT = SQLSTATE;
            leave while_loop;
            end while while_loop;
      close EMP1;
```
ITERATE

Iterate in REPEAT

```sql
set RESULT = '?????';
open EMP1;

repeat_loop:
  repeat
    fetch EMP1 into FNAME, LNAME;
    if (SQLSTATE = '00000')
      then
        set P_COUNT = P_COUNT + 1;
        iterate repeat_loop;
      end if;

    set RESULT = SQLSTATE;
    leave repeat_loop;
  until (9 = 0)
end repeat repeat_loop;
close EMP1;

Iterate in LOOP

```sql
set RESULT = '?????';
open EMP1;

loop_loop:
  loop
    fetch EMP1 into FNAME, LNAME;
    if (SQLSTATE = '00000')
      then
        set P_COUNT = P_COUNT + 1;
        iterate loop_loop;
      end if;

    set RESULT = SQLSTATE;
    leave loop_loop;
  end loop loop_loop;
close EMP1;
```

/*
 * Title       P_FNAME                   P_COUNT  RESULT
 * +------------+----------------+----------+--------
 * ++          ------+-----------+--------+
 */

call USER01.TITERATE1('TITERATE1','James ',0,'U')++
```
The LEAVE statement continues execution with the statement that immediately follows the specified labeled statement.

**Syntax**

```
LEAVE  stmt-label
```

**Parameters**

`stmt-label`

Specifies the begin label of a statement that contains the LEAVE statement, and identifies the statement that needs to be left.

**Usage**

*Statements that may be left:* The labeled statement referred in the LEAVE must be a LOOP, REPEAT, WHILE or compound statement that contains the LEAVE statement.
Example

```sql
set options command delimiter '++';
create procedure USER01.TLEAVE1
  ( TITLE     varchar(10) with default
  , P_FNAME   char(20)
  , P_COUNT   integer
  , RESULT    varchar(25)
  )
  external name TLEAVE1 language sql

Label_700:
  /*
  ** Count number of employees with equal Firstname
  */
begin not atomic
  declare FNAME   char(20);
  declare LNAME   varchar(20);
  declare EMP1 CURSOR FOR
    select EMP_FNAME, EMP_LNAME
    from DEMOEMPL.EMPLOYEE
    where EMP_FNAME = P_FNAME;
  open EMP1;
  fetch EMP1 into FNAME, LNAME;
  fetching_loop:
    loop
      if (SQLSTATE <> '00000')
        then leave fetching_loop;
      end if;
      set P_COUNT = P_COUNT + 1;
      fetch EMP1 into FNAME, LNAME;
    end loop fetching_loop;

  set RESULT = 'SQLSTATE: ' || SQLSTATE;
  close EMP1;
end
++
call USER01.TLEAVE1('TLEAVE1','Martin',0);
++

*+ TITLE P_FNAME P_COUNT RESULT
*+ ----- ------- -------- -------
*+ TLEAVE1 Martin 3 SQLSTATE: 02000
```
The LOOP statement repeats the execution of a statement or a group of statements.

Syntax

```
beg-label: LOOP procedure-statement ; END LOOP end-label
```

Parameters

- **beg-label**: Specifies a 1- through 32-character SQL identifier that labels the LOOP statement. The value must be different from any other label used in the compound statement if the LOOP statement is contained in a compound statement.

```
LOOP procedure-statement END LOOP
```

- Specifies a statement or group of statements that are repeatedly executed.

- **end-label**: Specifies an SQL identifier that labels the end of the LOOP statement. If specified, a **beg-label** must also have been specified and both labels must be equal.

Usage

How execution of a LOOP statement ends

To end the repeated execution of the procedure-statements contained in a LOOP statement, a LEAVE statement can be used or an exit handler can be driven.
Example

See the example for the LEAVE statement. The procedure USER01.TLOOP1, is similar to USER01.TLEAVE1 but it uses an exit handler to terminate the LOOP.

```sql
set options command delimiter '++';
create procedure USER01.TLOOP1
  ( TITLE     varchar(10) with default
    , P_FNAME   char(20)
    , P_COUNT   integer
    , RESULT    varchar(30)
  )
  EXTERNAL NAME TLOOP1 LANGUAGE SQL
Label_700:
/*
** Count number of employees with equal Firstname
*/
begin not atomic
  declare FNAME   char(20);
  declare LNAME   varchar(20);
  declare EMP1 CURSOR FOR
    Select EMP_FNAME, EMP_LNAME
    From DEMOEMPL.EMPLOYEE
    where EMP_FNAME = P_FNAME;
  declare exit handler for SQLEXCEPTION, SQLWARNING, NOT FOUND
    set RESULT = 'SQLSTATE: ' || SQLSTATE;
/*
** Count number of employees with equal Firstname
*/
  open EMP1;
  fetch EMP1 into FNAME, LNAME;

  fetching_loop:
    loop
      set P_COUNT = P_COUNT + 1;
      fetch EMP1 into FNAME, LNAME;
    end loop fetching_loop;
end
++
commit++
set options command delimiter default++
call USER01.TLOOP1('TLOOP1','Martin ',0,'U');
++
++ TITLE     P_FNAME     P_COUNT
++ ------     -------     ------
++ TLOOP1     Martin     3
++
++ RESULT
++ ------
```
The REPEAT statement repeats the execution of a statement or a group of statements until a condition is met.

**Syntax**

```
REPEAT
  beg-label:
  procedure-statement;
UNTIL search-condition
END REPEAT
  end-label
```

**Parameters**

*beg-label:*

Specifies a 1- through 32-character SQL identifier that labels the REPEAT statement. The value must be different from any other label used in the compound statement if the REPEAT statement is contained in a compound statement.

*REPEAT procedure-statement*

Specifies the statement or group of statements that are repeatedly executed.

*UNTIL search-condition*

Specifies the search condition that is evaluated after each iteration. If the outcome is true, the statement following the REPEAT statement is executed. Otherwise, a new iteration starts.

*end-label*

Specifies an SQL identifier that labels the end of the REPEAT statement. If specified, a beg-label must also have been specified and both labels must be equal.
Example

set options command delimiter '++';
create procedure USER01.TREPEAT1
    ( TITLE     varchar(10) with default
      , P_FNAME   char(20)
      , P_COUNT   integer
      , RESULT    varchar(25)
    )
    EXTERNAL NAME TREPEAT1 LANGUAGE SQL
Label_700:
    /*
    ** Count number of employees with equal First name using REPEAT
    */
    begin not atomic
      declare FNAME   char(20);
      declare LNAME   varchar(20);
      declare EMP1 CURSOR FOR
        Select EMP_FNAME, EMP_LNAME
        From DEMOEMPL.EMPLOYEE
        where EMP_FNAME = P_FNAME;
      open EMP1;
      fetching_loop:
        repeat
          fetch EMP1 into FNAME, LNAME;
          if (SQLSTATE = '00000')
            then set P_COUNT = P_COUNT + 1;
          end if;
          until SQLSTATE <> '00000'
        end repeat fetching_loop;
      set RESULT = 'SQLSTATE: ' || SQLSTATE;
      close EMP1;
    end
    ++
    commit++
    set options command delimiter default++
call USER01.TREPEAT1('TREPEAT1','Martin',0,'U');
*+
  *+ TITLE       P_FNAME                   P_COUNT  RESULT
  *+ ......       ........               -------  -------
  *+ TREPEAT1     Martin                   3        SQLSTATE: 02000
The RESIGNAL statement ressignals an SQL event or exception condition in a handler for the next higher level scope.

Syntax

```
RESIGNAL
    SQLSTATE  VALUE 'sqlstate'
      condition-name

SET MESSAGE_TEXT = simple-value-specification
```

Parameters

'sqlstate'

Specifies the value for SQLSTATE that is to be ressignaled. 'Sqlstate' is a 5-character string-literal that consists of only digits (0-9) and capital alphabetic characters (A-Z). 'Sqlstate' cannot be '00000', the value of SQLSTATE for successful completion.

condition-name

Specifies the name of a condition whose SQLSTATE value is to be ressignaled. Condition-name must identify a condition defined by a condition-declaration in a compound-statement containing the RESIGNAL statement. If more than one such condition-declaration has the specified condition-name, the one with the innermost scope is raised.

simple-value-specification

Specifies a character value to be added to the information item MESSAGE-TEXT. simple-value-specification must have a character data type.
Usage

Propagating the SQL Condition

The RESIGNAL statement can only be used in a handler to propagate an SQL condition to the scope that encloses the exception handler's scope. If the RESIGNAL is issued in a handler of a top level compound statement, control returns to the invoker of the SQL-invoked routine.

FLOW of CONTROL

If in the outer scope a handler exists for the raised exception or SQL event, the handler acquires control. After execution of the handler, control returns as with any other statement that causes a handler to activate.

SQLSTATE

There are no restrictions on the values that can be set for SQLSTATE, other than compliance with the syntactic rules for SQLSTATE values. We recommend using values in accordance with the classification of SQLSTATE values.

MESSAGE_TEXT

This is an information item of character type with a length of 80.
Example

set options command delimiter '++';
create procedure USER01.RESIGNAL1
  ( TITLE     varchar(10) with default
    , RESULT    varchar(120)
  )
  EXTERNAL NAME RESIGNA1 LANGUAGE SQL
Label_400:
  /*
  ** Resignal show case
  */
  begin not atomic
    declare DEAD_LOCK condition for SQLSTATE '12000';
    declare NOT_FOUND condition for SQLSTATE '02000';
    declare exit handler for NOT_FOUND
      begin not atomic
        set RESULT = RESULT || ' Not Found';
        resignal SQLSTATE '38607';
      end;
    set RESULT = 'Signal trace:';
    signal NOT_FOUND;
  end label_400
++
commit;
set options command delimiter++
call user01.resignal1('Signal');

*+ Status = -4    SQLSTATE = 38000    Messages follow:
*+ DB001075 C-4M321: Procedure RESIGNA1 exception 38607

RETURN

The RETURN statement returns a value for an SQL function. As an extension to the SQL standard, a RETURN without parameters can also be used to exit a compound statement.

Syntax

```
| RETURN | NULL | value-expression |
```
Parameters

NULL

Specifies that the function return value is NULL.

value-expression

Specifies the function return value.

Usage

Compatible Data Types

The data type of the value-expression and the data type of the function return value named in the CREATE FUNCTION statement must be compatible for assignment.

Example

For an example, see CREATE FUNCTION.

SET Assignment

The SET Assignment statement assigns values to parameters and variables used in SQL routines.

Syntax

```sql
SET local-variable = value-expression
```

Parameters

local-variable

Identifies the local variable that is the target of the SET assignment statement. Local-variable must be the name of a local variable defined within the compound statement containing the SET statement.

routine-parameter

Identifies the SQL routine parameter that is the target of the SET assignment statement. Routine-parameter must be the name of a parameter of the routine containing the SET assignment statement.

value-expression

Specifies the value to be assigned to the target of the SET assignment statement.

NULL

Specifies that the null value is to be assigned to the target of the SET assignment statement.
Usage

Valid assignments

The rules for assignment are provided in Comparison, Assignment, Arithmetic, and Concatenation Operations.

Example

The procedure TSET3 creates a combined, edited name from a given first and last name. If the first or last name is null, or if the length of the last name is 0, the null value is returned for the edited name.

```sql
set options command delimiter '++';
create procedure SQLROUT.TSET3
  ( P_FNAME   varchar(20)
  , P_LNAME   varchar(20)
  , P_NAME   varchar(41)
  )
  EXTERNAL NAME TSET3 LANGUAGE SQL

  /*
  ** Return an edited name from the given Firstname and Lastname
  */
  if (LENGTH(P_LNAME) <= 0)
    then set P_NAME = null;
  else set P_NAME = trim(P_FNAME)  || ' ' || trim(P_LNAME) ;
  end if

  set options command delimiter default++

  call SQLROUT.TSET3('James    ', 'Last   ');
  *+ 
  *+ P_FNAME               P_LNAME 
  *+ -------               ------- 
  *+ James                 Last  
  *+ P_NAME 
  *+ ------- 
  *+ James Last 

  call SQLROUT.TSET3('James    ', '');
  *+ 
  *+ P_FNAME               P_LNAME 
  *+ -------               ------- 
  *+ James                 
  *+ P_NAME 
  *+ ------- 
  *+ <null>
```
SIGNAL

The SIGNAL statement raises and signals an SQL event or exception condition.

Syntax

```plaintext
SIGNAL SQLSTATE 'sqlstate' 
```

```
SET MESSAGE_TEXT = simple-value-specification
```

Parameters

'sqlstate'

Specifies the value for SQLSTATE that is to be signaled. 'sqlstate' is a 5-character string-literal value that consists of only digits (0-9) and capital alphabetic characters (A-Z). 'Sqlstate' cannot be '00000', the value of SQLSTATE for successful completion.

condition-name

Specifies the name of a condition whose SQLSTATE value is to be signaled. `Condition-name` must identify a condition defined by a condition declaration in a compound statement containing the SIGNAL statement. If more than one such condition declaration has the specified condition name, the one with the innermost scope is raised.

simple-value-specification

Specifies a character value to be added to the information item MESSAGE-TEXT. `simple-value-specification` must have a character data type.

Usage

FLOW of CONTROL

If a handler exists for the raised exception or SQL event, the handler acquires control. After execution of the handler, control returns as with any other statement that causes activation of a handler.

If no handler is activated, control goes to the end of the compound statement that contains the signal. If the signal is not in a compound statement of an exit handler, control returns to the invoker of the SQL routine. Otherwise, it returns to the statement after the SIGNAL statement, just as if a continue handler had been activated.
SQLSTATE

There are no restrictions on the values that can be set for SQLSTATE, other than compliance with the syntactic rules for SQLSTATE values. We recommend that values are used in accordance with the classification of SQLSTATE values.

**Note:** For more information, see SQLSTATE Values.

MESSAGE_TEXT

This is an information item of character type with a length of 80.
Example

set options command delimiter '++';
create procedure USER01.TSIGNAL5
  ( TITLE     varchar(10) with default,
    RESULT    varchar(120)
  )
  EXTERNAL NAME TSIGNAL5 LANGUAGE SQL
Label_400:
  /*
   ** Trace execution of consecutive signal statements
   */
  begin not atomic
    declare DEAD_LOCK condition for SQLSTATE '12000';
    declare NOT_FOUND condition for SQLSTATE '02000';
    declare continue HANDLER for SQLWARNING
      LABEL_9999:
        begin not atomic
          set RESULT = RESULT || ' Sqlwarning';
          end;
    declare continue handler for SQLEXCEPTION
      Label_8888:
        begin not atomic
          set RESULT = RESULT || ' Sqlexception';
          end;
    declare continue handler for SQLSTATE '23800'
      set RESULT = RESULT || ' 23800';
    declare continue handler for DEAD_LOCK
      LABEL_6666:
        begin not atomic
          set RESULT = RESULT || ' Deadlocked';
          end;
    declare continue handler for NOT FOUND
      set RESULT = RESULT || ' Not Found';
    set RESULT = 'Signal trace:';
    signal SQLSTATE '23800';
    signal NOT_FOUND;
    signal SQLSTATE '01200';
    signal SQLSTATE '72300';
    signal DEAD_LOCK;
  end label_400
++
commit++
set options command delimiter default++
call user01.tsignal5('Signal');
WHILE

The WHILE statement repeats the execution of a statement or a group of statements while a condition is met.

**Syntax**

```
 beg-label:       WHILE  search-condition  DO 

 procedure-statement  ;  END WHILE  end-label
```

**Parameters**

**beg-label:**

Specifies a 1- through 32-character SQL identifier that labels the WHILE statement. The value must be different from any other label used in the compound statement if the WHILE statement is contained in a compound statement.

**WHILE search-condition**

Specifies the search condition to be evaluated. If the outcome is false, the statement after the WHILE statement is executed. Otherwise, an iteration of the group of statements enclosed by DO and END WHILE is started.

**DO procedure-statement END WHILE**

Specifies the statement or group of statements that are repeatedly executed.

**end-label**

Specifies an SQL identifier that labels the end of the WHILE statement. If specified, a *beg-label* must also have been specified and both labels must be equal.
Example

set options command delimiter '++';
create procedure USER01.TWHILE2
    ( TITLE     varchar(10) with default
      , P_FNAME   char(20)
      , P_COUNT   integer
    )
    EXTERNAL NAME TWHILE2 LANGUAGE SQL
Label_700:
begin not atomic
    /*
    ** Count number of employees with equal first name
    */
    declare FNAME   char(20);
    declare LNAME   varchar(20);
    declare EMP1 CURSOR FOR
        Select EMP_FNAME, EMP_LNAME
        From DEMOEMPL.EMPLOYEE
        where EMP_FNAME = P_FNAME;
    set P_COUNT = 0;
    open EMP1;
    fetch EMP1 into FNAME, LNAME;
    fetching_loop_non_SQL:
    while (SQLSTATE = '00000')
        do
            set P_COUNT = P_COUNT + 1;
            fetch EMP1 into FNAME, LNAME;
        end while fetching_loop_non_SQL;
    close EMP1;
end
++
commit++
set options command delimiter default++
call USER01.TWHILE2('TWHILE2','Martin ');
;
*+ TITLE    P_FNAME    P_COUNT
*+ ------    --------    -------
*+ TWHILE2  Martin      3
Chapter 10: Accessing Non-SQL-Defined Databases

This section contains the following topics:

- Correspondence between SQL and Non-SQL-defined Entities (see page 599)
- SQL Schema Considerations (see page 604)
- SQL DML Statements Operating on Non-SQL-defined Records (see page 606)
- SQL Access to Non-SQL Databases (see page 609)
- Expansion of Extended-search Condition (see page 613)
- Expansion of Set-specification Statement (see page 613)

Correspondence between SQL and Non-SQL-defined Entities

Procedure

Define an SQL SCHEMA using the CREATE SCHEMA statement containing a FOR NONSQL SCHEMA parameter. This causes a logical relationship to be established between the two schemas.

The records defined in the non-SQL schema can now be accessed as tables in SQL DML and CREATE VIEW statements. Each record element is represented as a column except as noted under Record Structure Considerations.

Table Names

The table name used when referring to a non-SQL defined record is always the name specified in the ADD RECORD statement of the non-SQL schema definition. If hyphens appear in the record name, the table name must be enclosed in double quotation marks.

Column Names

The column name used when referring to a record element depends on whether a synonym for LANGUAGE SQL has been defined for the record:

- If such a synonym has been defined, the column name is the element synonym name associated with the SQL record synonym.
- If no SQL synonym has been defined for the record, the column names are based on the names of record elements either defined in the schema RECORD statement or associated with the version of the record whose structure is shared by the schema record.

In either case, column names are transformed by replacing hyphens with underscores.
Elements occurring a fixed number of times are represented by multiple columns whose names are constructed from the element name appended with an underscore (_) and an occurrence count. For example:

02  MONTHLY-BUDGET OCCURS 12 TIMES ... 

is represented as 12 columns whose names are:

MONTHLY_BUDGET_01
MONTHLY_BUDGET_02 etc.

The length of the suffix is one greater than the number of digits in the number of occurrences of the element.

Up to three levels of nesting are supported. For example:

02  ANNUAL-BUDGET OCCURS 4 TIMES.
    03  MONTHLY-BUDGET OCCURS 12 TIMES ... 

is represented as 48 columns whose names are:

MONTHLY_BUDGET_1_01
MONTHLY_BUDGET_1_02

.

MONTHLY_BUDGET_1_12
MONTHLY_BUDGET_2_01
MONTHLY_BUDGET_2_02 etc.

In constructing column names for multiply-occurring elements, if the length of the resulting name exceeds 32 characters, the record is not accessible through SQL. To overcome this, define a synonym for LANGUAGE SQL where the affected element names are shortened.
Record Structure Considerations

Not all record elements can be referred to as columns and, in some cases, a single record element is represented by multiple columns.

Elements, other than those whose usage is BIT, are handled as follows:

- Group elements are not represented by columns.
- FILLERs (elements with names of 'FIL nnnn') are not represented by columns.
- Level 88 elements (condition names) are not represented by columns.
- Redefining elements and elements subordinate to a redefining group are not represented by columns.
- Elements occurring a fixed number of times and elements subordinate to a group occurring a fixed number of times are represented by multiple columns, one for each occurrence of the element.
- Elements occurring a variable number of times and elements subordinate to a group occurring a variable number of times are not represented by columns.
- All other elements in the record are represented by columns.

USAGE BIT

Elements whose usage is BIT are not represented by columns except as noted following:

- Group elements where all subordinate elements have a usage of BIT and which start on a byte boundary are represented by columns with a data type of BINARY. The length of the column is the length in bytes from the start of the group element to the start of the next element at the same level which begins on a byte boundary. If groups are nested within groups, the group element with the lowest level number where all subordinate elements are BITs is the element represented by a column. Intervening and subordinate elements are not represented by columns.
- BIT elements occurring a fixed number of times and beginning on a byte boundary are represented by columns with a data type of BINARY. The length of the column is the length in bytes from the start of the element to the start of the next element at the same level which also begins on a byte boundary. Intervening elements are not represented by columns.
- Other BIT elements which begin on a byte boundary are represented by columns with a data type of BINARY. The length of the column is the length in bytes from the start of the element to the start of the next element at the same level which also begins on a byte boundary. Intervening elements are not represented by columns.
Data Type of Columns

The data type of a column representing a record element is derived from the picture and usage of the element:

<table>
<thead>
<tr>
<th>Picture and usage</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC X(n) usage DISPLAY</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>PIC A(n) usage DISPLAY</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>Numeric edited</td>
<td>CHAR(l), l=byte length</td>
</tr>
<tr>
<td>External floating point</td>
<td>CHAR(l), l=byte length</td>
</tr>
<tr>
<td>PIC G(n) usage DISPLAY</td>
<td>GRAPHIC(n)</td>
</tr>
<tr>
<td>PIC S9(p)V9(s) usage DISPLAY</td>
<td>NUMERIC(p-s,s)</td>
</tr>
<tr>
<td>PIC SP..9(p) usage DISPLAY &amp; sub3.</td>
<td>NUMERIC(p, p)</td>
</tr>
<tr>
<td>PIC S9(p)P.. usage DISPLAY &amp; sub3.</td>
<td>NUMERIC(p, 0)</td>
</tr>
<tr>
<td>PIC 9(p-s)V9(s) usage DISPLAY</td>
<td>UNSIGNED NUMERIC(p, s)</td>
</tr>
<tr>
<td>PIC P..9(p) usage DISPLAY &amp; sub3.</td>
<td>UNSIGNED NUMERIC(p, p)</td>
</tr>
<tr>
<td>PIC 9(p)P.. usage DISPLAY &amp; sub3.</td>
<td>UNSIGNED NUMERIC(p, 0)</td>
</tr>
<tr>
<td>PIC S9(p-s)V9(s) usage COMP-3</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>PIC SP..9(p) usage COMP-3 &amp; sub3.</td>
<td>DECIMAL(p, p)</td>
</tr>
<tr>
<td>PIC S9(p)P.. usage COMP-3 &amp; sub3.</td>
<td>DECIMAL(p, 0)</td>
</tr>
<tr>
<td>PIC 9(p-s)V9(s) usage COMP-3</td>
<td>UNSIGNED DECIMAL(p, s)</td>
</tr>
<tr>
<td>PIC P..9(p) usage COMP-3 &amp; sub3.</td>
<td>UNSIGNED DECIMAL(p, p)</td>
</tr>
<tr>
<td>PIC 9(p)P.. usage COMP-3 &amp; sub3.</td>
<td>UNSIGNED DECIMAL(p, 0)</td>
</tr>
<tr>
<td>PIC S9(n), n&lt;5 usage COMP &amp; sub4.</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>PIC S9(n), 4&lt;n&lt;10 usage COMP &amp; sub4.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>PIC S9(n), 9&lt;n usage COMP &amp; sub4.</td>
<td>LONGINT</td>
</tr>
<tr>
<td>PIC 9(n) usage COMP &amp; sub4.</td>
<td>BINARY(l), l=byte length</td>
</tr>
<tr>
<td>PIC X(n) usage BIT</td>
<td>BINARY(l), l=byte length</td>
</tr>
<tr>
<td>USAGE POINTER</td>
<td>BINARY(4)</td>
</tr>
<tr>
<td>USAGE COMP-1</td>
<td>REAL</td>
</tr>
<tr>
<td>USAGE COMP-2</td>
<td>DOUBLE PRECISION</td>
</tr>
</tbody>
</table>
Correspondence between SQL and Non-SQL-defined Entities

Chapter 10: Accessing Non-SQL-Defined Databases

<table>
<thead>
<tr>
<th>Picture and usage</th>
<th>Data type</th>
</tr>
</thead>
</table>

Note:

1. Numeric edited includes any element whose usage is DISPLAY and:

Whose picture contains any of the editing symbols: + - Z B 0 S CR DB . , *

Whose picture clause contains only the symbols: 9 (n) V S P but whose element description also includes the SIGN LEADING or SEPARATE CHARACTER specification

2. External floating point includes any element whose usage is DISPLAY and whose picture is: +/- mantissa E +/- exponent

3. The scaling character "P" in a picture clause is ignored in value representations of associated columns. This has the effect of representing values of such columns as a power of 10 greater than or smaller than their actual value. For example, if an element is described as PIC S9(5)PPP, a value of 123000 is represented in SQL as 123. If an element is described as PIC SPPP9(5), a value of .000123 is represented in SQL as .123.

4. Computational elements also include those whose USAGE is BINARY and COMP-4. If the picture of a computational item includes an implied decimal point, it is ignored in determining the data type of the column. This has the effect of representing values of such columns as a power of 10 greater than their actual values. For example, if an element is described as PIC S9(5)V99 USAGE COMP, a value of 123.56 is represented in SQL as 12345. 😄
SQL Schema Considerations

SQL Schemas and Non-SQL-defined Schemas

Typically, one SQL schema is defined for each non-SQL schema which describes data to be accessed through SQL. The SQL schema definition specifies the name of the segment or database containing the data described by the non-SQL schema.

However, a single non-SQL schema may describe multiple physical databases. In this case, one SQL schema can be used to access any of the physical implementations, or a separate SQL schema can be defined for each. To make the SQL schema independent of the physical implementation, omit the DBNAME specification from the schema definition.

If No DBNAME is Specified

If no DBNAME is specified in the SQL schema definition, only one physical instance of the non-SQL-defined database can be accessed within a single SQL transaction. The data accessed at runtime is determined by the database name to which your SQL session is connected. The database name must include the segments containing the data to be accessed.

DBNAME Specification and Access Modules

The way you choose to associate the non-SQL-defined schema with an SQL-defined schema has an impact on access modules:

- If the SQL schema does not contain a DBNAME specification, a single access module can be used against any of the physical databases because the application can specify the appropriate database name on a CONNECT statement (or the user can specify it in a DCUF or SYSIDMS DICTNAME parameter).

- If the SQL schema contains a DBNAME specification, each physical implementation must have its own set of access modules and the application must specify which one to use by issuing a SET ACCESS MODULE statement.

Restriction

If two or more non-SQL-defined schemas describe the same physical data, it should be accessed under only one SQL schema within a transaction. For example, if the same physical employee information is described in two non-SQL-defined schemas referenced by the two SQL schemas HR and MFG, the employee records should either be accessed as HR tables or MFG tables, but not both.
**Preventing Unpredictable Results**

Unpredictable results occur (including possible database corruption) if the above restriction is violated.

To prevent unpredictable results, grant access to only one table for each non-SQL-defined record. In the above example, grant access to either the HR.EMPLOYEE table or the MFG.EMPLOYEE table but not both.

**Definition Changes and Access Modules**

Changes made to the non-SQL-defined schema do not cause access modules to be automatically recompiled (as is the case for SQL-defined entities). This is because there are no synchronization stamps that CA IDMS can use to detect definition changes for non-SQL-defined records. It is the DBA's responsibility to recompile the access modules when changes are made to the non-SQL-defined schema.

If an automatic recompile of the access module is desired, the RCM must be recreated through a recompile of the program to change the RCM synchronization stamps. Then when the access module is run, an access module recompile is triggered.

Other changes that may necessitate access module recompilation are:

- Changing the dictionary, name, or version number of the non-SQL-defined schema associated with an SQL-defined schema
- Changing the DBNAME parameter associated with the SQL-defined schema
To recompile affected access modules, use the ALTER ACCESS MODULE statement with the REPLACE ALL option. To determine which access modules are affected, query the SYSTEM.AM and SYSTEM.AMDEP tables in the dictionary.

**Note:** For more information, see ALTER ACCESS MODULE and SYSTEM Tables and SYSCA Views.

**Definition Changes and Views**

The deletion of records and changes made to the structure of records in a non-SQL-defined schema may necessitate the dropping and recreating of views referencing the non-SQL-defined tables representing those records. The following changes invalidate referencing views:

■ Removal of the record from the schema
■ Addition of record elements (record elements added to the end of the record structure invalidate only those views based on SELECT *)
■ Re-ordering of record elements within the record
■ Changing of an element’s picture or usage

To determine which views are impacted by such changes, query the SYSCA.VIEWDEP table.

**Note:** For more information, see SYSTEM Tables and SYSCA Views.

## SQL DML Statements Operating on Non-SQL-defined Records

### INSERT

CA IDMS allows INSERT statements where the target table represents a non-SQL-defined record only if all sets with a membership option of AUTOMATIC in which the record participates as a member have been defined with a primary/foreign key declaration.

**Note:** For more information about primary and foreign keys in set definitions, see the *CA IDMS Database Administration Guide*.

Additionally, you cannot include the control field of an OCCURS DEPENDING ON structure in the insert column list. On an INSERT, its value is automatically set to 0.
Effect of INSERT on a Record

- Causes an occurrence of the record represented by the table named in the INTO parameter to be stored on the database.
- Columns whose values are not supplied on the insert are given standard default values according to their data types (that is, 0 for numeric, spaces for character and binary zeros for binary).
- Has the following effect on system indexes defined on the record:
  - Connects the record into every such index defined as AUTOMATIC.
  - Does not connect the record into any index defined as MANUAL.
- Connects the record into every set for which the values of all columns representing foreign key fields of the set relationship are not null. The set occurrence to which the record is connected is that owned by the occurrence of the owner record whose primary key value matches the member's foreign key value.
- Returns an error if:
  - In attempting to connect the record into a set, no owner record occurrence can be found with a matching primary key value.
  - A null value has been specified for any column other than one representing a nullable foreign key of some set where the record participates as a member.
  - An invalid data value is detected during the operation.
  - The operation attempts to store a duplicate row when duplicates are not allowed.

UPDATE

UPDATE statements where the target table is a non-SQL-defined database record are allowed. Successful execution of an update operation, however, depends on both the definition options chosen and the current state of the database.

The control field of an OCCURS DEPENDING ON structure cannot be updated.
Effect of UPDATE on a Record

- Causes one or more occurrences of the record represented by the table to be modified.
- Only fields represented by columns named in the SET parameter of the UPDATE statement for which the update value is not NULL are changed.
- Has the following effect on system indexes defined on the record:
  - If a record occurrence being updated is connected to such an index and one or more index key fields are changed, the index is updated.
  - If a record occurrence being updated is not connected to such an index, the index is not updated.
- Has the following effect on sets where the record is a member:
  - If one or more foreign key fields of such a set are changed or are set to NULL, the following operations are performed:
    - The record occurrence is disconnected from its current owner if it participates as a member of the set.
    - The record occurrence is connected to a (new) owner if the value of all foreign key fields are not null and an occurrence of the owner record can be found with a matching primary key.
    - The record's set membership with owner records whose keys are defined in the member record as foreign keys will be affected regardless of the membership options of the set.
  - If one or more sort key fields of such a set are changed, and the record occurrence is a member of the set, the set occurrence is updated to maintain correct ordering.
- Returns an error if:
  - In attempting to connect the record into a set, no owner record occurrence can be found with a matching primary key value.
  - The record being modified is an owner of a non-empty set that was defined with a primary/foreign key declaration and one or more of the primary key fields are changed.
  - A null value is specified for a column other than one representing a nullable foreign key field of some set where the record participates as a member.
  - An invalid data value is detected during the operation.
  - The operation attempts to store a duplicate row when duplicates are not allowed.
DELETE

DELETE statements where the target table represents a non-SQL-defined record, are allowed. Successful execution of such a statement depends both on the definition options chosen and the current state of the database.

Effect of DELETE on a Record

- Causes one or more occurrences of the record represented by the table to be erased from the database
- Disconnects a record occurrence from all indexes where it participates
- Disconnects a record occurrence from all sets where it participates as a member
- Returns an error if the record occurrence being erased participates as an owner in one or more non-empty sets

SELECT

SELECT statements where one or more tables named in the FROM parameter represent non-SQL defined records are always allowed.

Column values are established as follows:

- If the column represents a nullable foreign key field, its value is:
  - The value of the field in the record occurrence, if the record occurrence is a member of at least one set where the field is a foreign key field
  - Null if the record occurrence is not a member of any set where the field is a foreign key field
- Otherwise, the value of the column is the value of the record element it represents

An exception is raised if a value in the result table is null and an indicator variable is not specified for the host-variable to which the value is to be returned.

SQL Access to Non-SQL Databases

This section provides a review of the transformations used by the SQL engine while reading the definitions of non-SQL record types. When SQL is used to access non-SQL record types, the entity names coded in the SQL syntax must follow the conventions described next.

Note: For more information about defining and using table procedures to process non-SQL-defined data in a relational way even if the data does not conform to the rules for such access, see Defining and Using Table Procedures.
SQL Schemas for Non-SQL Databases

SQL tables are referenced in SQL DML statements by coding the table name preceded by a schema name qualifier. For example, in `SELECT * FROM DEMOSCH.SAMPLE`, `SAMPLE` is the table name and `DEMOSCH` is the SQL schema where it is defined. The combination of schema name and table name allows the SQL compiler to look up the definition of the table in the SQL catalog.

To access a non-SQL record type from an SQL statement, you must code the record name in the same manner used for SQL tables. An SQL schema which maps onto the corresponding non-SQL schema must be defined in the SQL catalog. This SQL schema name is used to qualify all subsequent references to non-SQL record types in SQL DML statements. For example,

```
CREATE SCHEMA SQLNET FOR NONSQL SCHEMA PRODICT.CUSTSCHM;
SELECT * FROM SQLNET."ORDER-NET";
```

Non-SQL Record and Set Name Transformations

Because hyphen is the subtraction operator in SQL, non-SQL record names containing embedded hyphens must be delimited by double quotes (for example, "CUST-REc-123"). Any non-SQL set names containing embedded hyphens must be delimited by double quotes before they can be used in an SQL statement (for example, "CUST-ORDER").

Non-SQL Element Name Transformations

Unlike hyphens embedded in record and set names, hyphens embedded in non-SQL element names are automatically transformed to underscores _ during the definition loading phase of the SQL compiler. So, to access the CUST-NUMBER element in a non-SQL record type, you must code CUST_NUMBER in an SQL statement.

When a FOR LANGUAGE SQL synonym is defined for a non-SQL record type, the element synonyms are used for all SQL access. SQL synonyms are only used for element names. Defining SQL synonyms for non-SQL record types is sometimes the only way to overcome column name limitations within SQL.

Some non-SQL element names don't make satisfactory SQL column names, even after the hyphen-to-underscore transformation. For example, if a non-SQL element name starts with a numeric character, the double quote delimiter must again be used (123-ORD-NUM would be accessed using "123_ORD_NUM" in an SQL statement).
Group elements, redefines elements, FILLERS and OCCURS ... DEPENDING ON elements are simply not available for access by SQL. The SQL user views these elements as not being defined in the non-SQL record type. However, the subordinate elements of a group definition are available, as are the base elements to which a REDEFINES is directed.

Though OCCURS ... DEPENDING ON declarations are not available for SQL access, fixed OCCURS definitions are made available. The SQL user’s perception of a fixed OCCURS element is that there is one column for each occurrence of the element. The name which is used to access each such occurrence is the original element name followed by an underscore and an occurrence number to make the column name distinct. If the element is declared with nested OCCURS clauses, the corresponding column names contain one underscore and one occurrence number for each “dimension” of the OCCURS declaration. For example, the element definition BUD-AMT OCCURS 12 TIMES generates the following column names: BUD_AMT_01, BUD_AMT_02, BUD_AMT_03, ..., BUD_AMT_12.

In the preceding example, the occurrence number appended to the column name is made large enough to hold the largest subscript from the corresponding element definition. If the base element name in combination with the appended occurrence information makes the generated column names larger than 32 characters, you receive an error when the SQL statement is compiled. In this situation, you must define an SQL synonym for the non-SQL record type. The synonym element names must be short enough so the appended occurrence information will not make the resulting column names larger than 32 characters.

Although the SQL implementation in CA IDMS allows 32 character column names, other SQL implementations restrict column names to 18 characters. In particular, some ODBC client software may require you to use SQL synonyms for non-SQL record types to limit the size of the transformed column name to 18 characters.
Definition Anomalies of Non-SQL Record Types

Certain definition anomalies of non-SQL record types can result in errors during attempts to access them with SQL. These anomalies pertain to the definition of CALC keys, system-owned index set keys, and user-owned sorted set keys. They result in a DB002024 error in Release 12.0 or a DB002038 error in later releases.

The DB002038 message includes both the set name and record type in question. The DB002024 message only includes the set name. The DB002024 message presents a problem if a CALC definition is the cause of the error. CALC is the set name, and if there are several CALC records involved in the SQL statement, or if you are compiling an access module with references to numerous CALC records, you may have to examine the definitions of all CALC records to locate the problem.

Another characteristic of the CA IDMS SQL engine can further complicate the process of finding such errors. The SQL compiler loads the definitions of all SQL tables and non-SQL record types explicitly referenced by the SQL statements being compiled. However, it also loads the definitions of the non-SQL record types which (through non-SQL set definitions) either own or are owned by the records which are explicitly referenced in the SQL statements, so that set-based access strategies can be considered when it optimizes each statement. This may result in a 2024 or 2038 error being generated for a record type which isn’t even referenced by the SQL being compiled.

These errors have only two known causes, both of them easily fixed:

1. The control key definition of the CALC, INDEX, or sorted set includes a FILLER element. To overcome this problem, simply modify the non-SQL record definition to assign a name other than FILLER to the element in question.

2. The control key definition incorporates subordinate elements of a group level REDEFINES, and these elements are smaller in size than the base element being redefined. For example:

   ```
   02 ELEM1 PIC X(8).
   02 ELEM1REDEF REDEFINES ELEME1.
       03 ELEM1A PIC S9(8) COMP.
       03 ELEM1B PIC S9(8) COMP.
   .
   .
   .
   ```

   An error occurs if ELEM1A and ELEM1B are used in the control key definition; since they are smaller than the element which they redefine even though together they are as large as ELEM1. The solution to this error is to change the redefining group, which contains the smallest subordinate elements, into the base element definition. This base definition should be used in the control key specification. In the previous example, ELEM1REDEF should be the base element definition, and ELEM1 should be coded so that it redefines ELEM1REDEF.
Expansion of Extended-search Condition

The parameters used in the expansion of Extended-search Condition specify criteria used to select rows from tables.

Syntax

```
Expansion of extended-search-condition
     ▶▶─┬─ search-condition ▶▶     └─ ▼─ AND ┬─ search-condition ┘
     ┌─▼─ set-specification ┤
     └─▼─ set-specification ┘
```

Parameters

- **search-condition**
  Specifies a search condition whose value must be true for the row or rows to be included in the result table.

- **set-specification**
  Specifies that only rows participating as owner and member in the named set be included in the result table. For expanded set-specification syntax, see Expansion of Set-specification Statement.

Usage

- **Evaluation**
  The full search condition is satisfied when the value of all its operands are true. It is not satisfied when the value of any of its operands is either false or unknown.

- **Order of Evaluation**
  CA IDMS effectively evaluates from left to right after first evaluating each operand individually.

  You can use parentheses to override the default order of evaluation. Operands in parentheses are evaluated first.

Expansion of Set-specification Statement

The parameters used in the expansion of the Set-specification statement specify join criteria for tables representing owner and member records of a non-SQL-defined set.
Syntax

Expansion of set-specification

\[
\begin{align*}
FIRST \quad & set-name \\
LAST \quad & table-identifier \\
\text{alias} \quad & \end{align*}
\]

Parameters

*set-name*

Specifies the name of the set to be used as the test criteria.

Set-name must follow the rules for identifiers. If hyphens appear in the name, it must be enclosed in double quotes.

*table-identifier*

Specifies the name of a table representing either the owner or member of the set. Table-identifier must appear in the FROM parameter of the containing query specification or SELECT statement.

At most, two table names or aliases can qualify the set name and if both appear, one must identify the owner and the other must identify a member of the set.

*alias*

Specifies the alias assigned to the table representing the owner or member of the set.

If the table has been assigned an alias in the FROM parameter of the query specification or SELECT statement where set-specification appears, the alias and not the original table name must be used to qualify the set.

*FIRST*

Specifies only the first member record occurrence from each occurrence of set-name is returned in the join.

*LAST*

Specifies only the last member record occurrence from each occurrence of set-name is returned in the join. For chained sets, this command is only valid when the set linkage includes prior pointers.

**Note:** For more information about coding considerations and set linkage, see Chapter 3.3, Sets in the CA IDMS Navigational DML Programming Guide.
**Usage**

**Members without Foreign Keys**

Joining rows from different tables specified in a SELECT statement is usually done with comparison operations on column values. The most typical approach for SQL-defined tables is to use equal comparisons of the matching primary/foreign key columns of a referential constraint definition.

However, in a non-SQL-defined database, member records may not contain the key values of their owner records. For example, it is not necessary for the EMPLOYEE record to contain the department ID of its associated DEPARTMENT record if the relationship between the EMPLOYEE and DEPARTMENT records is represented by a set.

In such cases, column-based comparison cannot be used to process a join; instead, the SELECT statement must identify the set in the WHERE parameter using **set-specification**.

**Note:** A system-owned index is not a set joining two records; therefore, it cannot be used in the WHERE clause.

**Evaluation**

Two table rows satisfy the **set-specification** criteria if one is a member of the other in the named set. The value of the set-specification is considered **true** when this condition is satisfied and **false** otherwise.

The tables representing the owner and member records must appear in the FROM parameter of the containing query specification or SELECT statement.

Inclusion of the **FIRST** and **LAST** keywords renders **set-specification** false for all member occurrences except for the first or last, respectively. This additional syntax is included for use with sets with an inherent first-in-first-out or last-in-first-out organization.
Qualification Requirements

*Set-name* must be qualified under the following conditions:

- More than one member of a multi-member set has been named as a table in the preceding FROM parameter
- A table representing either the owner or a member has been assigned an alias in the preceding FROM parameter

Improved Efficiency of Join Operations

A non-SQL-defined member record can contain the value of its owner's key. However, unless the set definition in the non-SQL-defined schema identifies this as a foreign key, CA IDMS will not use the set in its access strategy when performing join operations. This may result in the choice of a less-than-optimal access strategy.

This can be overcome by using *set-specification* as part of the selection criteria.
Chapter 11: Defining and Using Table Procedures

This section contains the following topics:

- When to Use a Table Procedure (see page 617)
- Defining a Table Procedure (see page 618)
- Accessing a Table Procedure (see page 618)
- Table Procedure Parameters (see page 619)
- Writing a Table Procedure (see page 622)

When to Use a Table Procedure

You can use a table procedure to process non-SQL-defined data in a relational way even though the data does not conform to the rules established for such access.

Table procedures allow you to perform the following tasks:

- Make processing of complex structures, such as bills-of-materials, easier for an end user. Since the details of access to the underlying records or tables are encapsulated within the procedure, less knowledge is required on behalf of the user to process the data.

- Access non-SQL-defined data which does not conform to the rules associated with SQL. For example, a procedure can enable access to the variable portion of a record or support INSERT on a record without embedded foreign keys.

- Access all segments of a segmented database within a single SQL transaction. Since a table procedure can open more than one run unit or SQL session simultaneously, it can access the appropriate segment based on the value of an input parameter. If no appropriate segment key is available, it can access each segment serially.

- Access remote data. This enables a single SQL transaction to access data distributed across different nodes within an IDMS network while hiding the knowledge of the location of the data within the procedure itself.
Defining a Table Procedure

You define a table procedure using the CREATE TABLE PROCEDURE statement. In the following example, the table procedure ORGANIZATION is named and associated with schema EMP. The name of the program to be called to service a DML request against the procedure, EMPORG, is specified in the EXTERNAL NAME parameter. The parameters to be passed to and from the procedure are listed. Each parameter definition consists of a name and a data type.

```sql
CREATE TABLE PROCEDURE EMP.ORGANIZATION
  (TOP_KEY         UNSIGNED NUMERIC(4),
   LEVEL           SMALLINT,
   MGR_ID          UNSIGNED NUMERIC(4),
   MGR_LNAME       CHAR(25),
   EMP_ID          UNSIGNED NUMERIC(4),
   EMP_LNAME       CHAR(25),
   START_DATE      CHAR(10),
   STRUCTURE_CODE  CHAR(2))
EXTERNAL NAME EMPORG;
```

More Information

- For more information about syntax and parameters used in defining table procedures, see CREATE TABLE PROCEDURE.
- For more information and a detailed example about using a CREATE TABLE PROCEDURE, see Sample COBOL Table Procedure.

Accessing a Table Procedure

You access table procedures using SQL DML statements, as with base tables and views. You can reference table procedures any place where a table reference is permitted. Whether a specific table procedure supports an SQL operation depends on the user-written program. The program might, for example, support only retrieval operations and disallow INSERT, UPDATE and DELETE by returning an error if such an operation is attempted.

Access to a table procedure is controlled in the same way as for a table. GRANT and REVOKE statements on a resource type of TABLE are used to give and remove SELECT, INSERT, UPDATE, DELETE or DEFINE privileges on a table procedure.
Table Procedure Parameters

The parameters associated with a table procedure are treated like columns of a table. You can specify them within the column list of a SELECT or INSERT statement, the SET clause of an UPDATE statement, the ORDER BY clause of a SELECT statement or the search criteria of a WHERE clause. Additionally, you can specify parameter values within the table procedure reference itself.

**Column List, SET and ORDER BY References**

<table>
<thead>
<tr>
<th>Parameters referenced in</th>
<th>Specify</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column list of a SELECT statement</td>
<td>The columns that are returned to the invoking application</td>
</tr>
<tr>
<td>Column list of an INSERT statement</td>
<td>The columns having values that are supplied in the subsequent VALUES clause or query-specification</td>
</tr>
<tr>
<td>SET clause of an UPDATE statement</td>
<td>Columns which are assigned new values during the update operation</td>
</tr>
<tr>
<td>ORDER BY clause of a SELECT statement</td>
<td>The order the result rows of the procedure are returned to the requesting application</td>
</tr>
</tbody>
</table>

**WHERE Clause References**

WHERE clause references to parameters are used to filter the output of the table procedure. Each time a table procedure returns a set of output values, they are evaluated against the selection criteria specified in the WHERE clause and non-conforming "rows" are ignored.

The WHERE clause parameter references is used to pass input values to the table procedure. If the set of column values provided on a particular call to the table procedure matches the columns defined in a KEY on the same table procedure, the ESTIMATED ROWS and ESTIMATED I/Os specified for that KEY are used by the optimizer when the table procedure is joined with other tables or views. If you specify selection criteria in the form of an "=" comparison (that is, parameter = value), value is passed to the table procedure. Other types of selection criteria such as IN predicates or comparison predicates with > or < operators have no effect on the value of the parameters passed to the table procedure.
**Table Procedure Parameters**

**Note:** For more information about defining keys, see CREATE KEY.

Specifically, a reference to a parameter in a WHERE clause results in an input value being passed to the table procedure only if:

- It appears within an equality test
- The equality test is not combined with other predicates in the WHERE clause through the use of the OR operator
- The equality test is not preceded by the NOT operator

**WHERE Clause Parameter References**

The following examples illustrate how a parameter reference in a WHERE clause affects the value passed to the table procedure:

<table>
<thead>
<tr>
<th>WHERE clause</th>
<th>Parameter value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P1</td>
</tr>
<tr>
<td>P1 = 1</td>
<td>1</td>
</tr>
<tr>
<td>P1 &lt; 1</td>
<td>-null-</td>
</tr>
<tr>
<td>P1 = C1</td>
<td>C1</td>
</tr>
<tr>
<td>P1 = 2 AND P2 = 3</td>
<td>2</td>
</tr>
<tr>
<td>P1 = 2 AND P2 &gt; 3</td>
<td>2</td>
</tr>
<tr>
<td>P1 = 2 OR P2 = 3</td>
<td>-null-</td>
</tr>
<tr>
<td>P1 IN (2, 3, 8)</td>
<td>-null-</td>
</tr>
</tbody>
</table>

**Parameters in Table Procedure References**

You can also specify input parameter values within the table procedure reference itself. You can specify them on any reference to a table procedure except within an INSERT statement.

You specify parameter values supplied on a table procedure reference either positionally or as keyword/value pairs. You can combine them with WHERE clause references to form the set of values that are passed to the table procedure.

**Examples of Specifying Parameter Values**

The following example shows all the ways you can specify input parameter values.

```sql
SELECT * FROM EMP.ORG (MGR_ID = 7, EMP_ID = 127)
SELECT * FROM EMP.ORG (CAST(NULL AS NUM(4,0)),
                        CAST(NULL AS SMALLINT),
                        7,
                        CAST(NULL AS CHAR(25)), 127)
SELECT * FROM EMP.ORG (MGR_ID = 7) WHERE EMP_ID = 127
```
Difference Between Table Procedure Reference and WHERE Clause

One difference exists between parameter values specified through a WHERE clause and those specified within the table procedure reference. Parameter values specified within the table procedure reference are not used to filter the output from the table procedure as is the case for those specified within the WHERE clause. Parameter values specified within the table procedure reference affect only the input to the table procedure and not the output from the table procedure. Therefore, the above three select statements are equivalent only if the table procedure enforces the conditions specified through the table procedure reference.

Note: The sample table procedure in Sample COBOL Table Procedure does not enforce any criteria other than those that it uses to navigate the database.

Statistics and Optimization

Ideally, a table procedure should be written such that when certain sets of column values are provided (either through a WHERE clause or a procedure reference), the most efficient path can be used to access the data or join the table procedure to another data source.

If the set of column values provided on a particular call to the table procedure matches the columns defined in a KEY on the same table procedure, the ESTIMATED ROWS and ESTIMATED I/Os defined for that KEY are used during optimization; otherwise, if the ESTIMATED ROWS and ESTIMATED I/Os are defined for the table procedure, they are used. If the ESTIMATED ROWS and ESTIMATED I/Os are not specified, the optimizer defaults to 1000 and 100 respectively.

Normally, these statistics are used when the table procedure is the object a simple select statement. However, the optimizer also uses them internally when the table procedure is joined with other tables or views. If the nature of the join is such that the values for columns (defined as a keys) are passed to the table procedure, the statistics from the appropriate key are used when choosing an access plan.
Writing a Table Procedure

The program associated with a table procedure can be written in COBOL, PL/I, or Assembler. When called, the program is passed a fixed parameter list consisting of the parameters specified on the table procedure definition and additional parameters used for communication between CA IDMS and the table procedure.

Whenever a reference to a table procedure is made, CA IDMS calls the program associated with the table procedure to service the request. Part of the information passed to the table procedure is an indication of the type of action that the table procedure is to perform, such as "return the next result row" or "update the current row." The table procedure responds by performing the requested action or returning an error.

CA IDMS performs transaction and session management automatically in response to requests that the originating application issues. Changes to the database made by a table procedure are committed or rolled out together with other changes made within the SQL transaction. No special action is required of the table procedure to ensure this occurs.

The next section discusses writing a table procedure in detail.

For an example of a table procedure written in COBOL, see Sample Table Procedure Program.

Calling Arguments

The following sets of arguments are passed each time a table procedure is called:

- One argument for each of the parameters specified on the table procedure definition, passed in the order the parameters were declared
- One argument for each null indicator associated with a parameter specified in the table procedure definition, passed in the order the parameters were declared
- A set of common arguments used for communications between CA IDMS and the table procedure

The first two sets of arguments vary from one table procedure to another. They are used to pass selection criteria and insert/update values to the table procedure and result values from the table procedure.

The last set of arguments, shown in the next table, is the same for all table procedures.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result Indicator (fullword)</td>
<td>Not used</td>
</tr>
</tbody>
</table>
### Argument Contents

<table>
<thead>
<tr>
<th>Argument</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLSTATE (CHAR (5))</td>
<td>Status code returned by the table procedure:</td>
</tr>
<tr>
<td></td>
<td>- 00000—Indicates success</td>
</tr>
<tr>
<td></td>
<td>- 01Hxx—Indicates a warning</td>
</tr>
<tr>
<td></td>
<td>- 02000—Indicates no more rows</td>
</tr>
<tr>
<td></td>
<td>- 38xxx—Indicates an error</td>
</tr>
<tr>
<td>Table Procedure Name</td>
<td>Name of the table procedure</td>
</tr>
<tr>
<td>(CHAR (18))</td>
<td></td>
</tr>
<tr>
<td>Explicit Name</td>
<td>Not used</td>
</tr>
<tr>
<td>Message Text (CHAR (80))</td>
<td>Message text returned by the table procedure and displayed by CA IDMS in</td>
</tr>
<tr>
<td></td>
<td>the event of an error or warning</td>
</tr>
<tr>
<td>SQL Command Code</td>
<td>Code indicating the type of SQL request for which the table procedure is</td>
</tr>
<tr>
<td>(fullword)</td>
<td>being called. See Table Procedure Requests for a list of valid command</td>
</tr>
<tr>
<td></td>
<td>codes.</td>
</tr>
<tr>
<td>SQL Operation Code</td>
<td>Code indicating the type of request being made of the table procedure.</td>
</tr>
<tr>
<td>(fullword)</td>
<td>See Table Procedure Requests for a list of valid operation codes.</td>
</tr>
<tr>
<td>Instance Identifier</td>
<td>A unique value identifying the scan on which the table procedure is</td>
</tr>
<tr>
<td>(fullword)</td>
<td>to operate.</td>
</tr>
<tr>
<td>Local Work Area</td>
<td>A user-defined storage area maintained across calls to the table</td>
</tr>
<tr>
<td>(User-defined)</td>
<td>procedure.</td>
</tr>
<tr>
<td>Global Work Area</td>
<td>A user-defined storage area maintained across calls to the table</td>
</tr>
<tr>
<td>(User-defined)</td>
<td>procedure and capable of being shared by other SQL routines.</td>
</tr>
</tbody>
</table>

### Table Procedure Requests

Part of the information passed to the table procedure is the type of request being made. This information is conveyed in two parameters:

- The first parameter contains a code indicating the type of SQL statement for which the request is issued (for example, INSERT, OPEN). The table following "SQL command codes" lists valid SQL command codes.
- The second parameter is an internal operation code indicating the type of action expected of the table procedure. The table following "Operation codes" lists possible operation codes.
## SQL Command Codes

The following table lists SQL command code values.

<table>
<thead>
<tr>
<th>Command number</th>
<th>Statement type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Logical DDL</td>
</tr>
<tr>
<td>3</td>
<td>CLOSE</td>
</tr>
<tr>
<td>4</td>
<td>COMMIT</td>
</tr>
<tr>
<td>5</td>
<td>COMMIT continue</td>
</tr>
<tr>
<td>6</td>
<td>COMMIT release</td>
</tr>
<tr>
<td>7</td>
<td>CONNECT</td>
</tr>
<tr>
<td>8</td>
<td>DECLARE</td>
</tr>
<tr>
<td>9</td>
<td>DELETE searched</td>
</tr>
<tr>
<td>10</td>
<td>DELETE positioned</td>
</tr>
<tr>
<td>11</td>
<td>DESCRIBE</td>
</tr>
<tr>
<td>12</td>
<td>EXECUTE</td>
</tr>
<tr>
<td>13</td>
<td>TERMINATE</td>
</tr>
<tr>
<td>14</td>
<td>EXECUTE IMMEDIATE</td>
</tr>
<tr>
<td>16</td>
<td>FETCH</td>
</tr>
<tr>
<td>17</td>
<td>INSERT</td>
</tr>
<tr>
<td>18</td>
<td>LOCK TABLE</td>
</tr>
<tr>
<td>19</td>
<td>OPEN</td>
</tr>
<tr>
<td>20</td>
<td>PREPARE</td>
</tr>
<tr>
<td>21</td>
<td>RESUME</td>
</tr>
<tr>
<td>22</td>
<td>RELEASE</td>
</tr>
<tr>
<td>23</td>
<td>ROLLBACK</td>
</tr>
<tr>
<td>24</td>
<td>ROLLBACK release</td>
</tr>
<tr>
<td>25</td>
<td>SELECT</td>
</tr>
<tr>
<td>26</td>
<td>SET ACCESS MODE</td>
</tr>
<tr>
<td>27</td>
<td>SET TRANSACTION</td>
</tr>
<tr>
<td>28</td>
<td>SUSPEND</td>
</tr>
<tr>
<td>29</td>
<td>UPDATE searched</td>
</tr>
</tbody>
</table>
### Command number

<table>
<thead>
<tr>
<th>Command number</th>
<th>Statement type</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>UPDATE positioned</td>
</tr>
<tr>
<td>31</td>
<td>SET COMPILE</td>
</tr>
<tr>
<td>32</td>
<td>SET SESSION</td>
</tr>
</tbody>
</table>

### Operation Codes

The following table lists operation code values and their meanings:

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open Scan &amp; sub1.</td>
<td>Value 12</td>
<td>Requests the table procedure prepare itself for returning a set of result rows. Selection criteria specified in the WHERE clause or in the table procedure reference are passed as arguments to the table procedure.</td>
</tr>
<tr>
<td>Next Row</td>
<td>Value 16</td>
<td>Requests the table procedure return the next result row for the indicated scan. Next Row requests are repeated to return all the result rows for a scan. The table procedure can set an SQLSTATE value indicating that all rows have been returned.</td>
</tr>
<tr>
<td>Close Scan</td>
<td>Value 20</td>
<td>Informs the table procedure that no further Next Row requests will be issued for the scan. The table procedure may free resources in response to this request.</td>
</tr>
<tr>
<td>Update Row</td>
<td>Value 40</td>
<td>Requests the table procedure update the &quot;current&quot; row of the indicated scan using the values of the passed parameters as the update values. Update Row requests are issued in response to either searched or positioned UPDATE statements.</td>
</tr>
<tr>
<td>Delete Row</td>
<td>Value 36</td>
<td>Requests the table procedure delete the &quot;current&quot; row of the indicated scan. Delete Row requests are issued in response to either searched or positioned DELETE statements.</td>
</tr>
<tr>
<td>Insert Row</td>
<td>Value 32</td>
<td>Requests the table procedure insert a row into the database using the values of the passed parameters as the insert values.</td>
</tr>
<tr>
<td>Suspend Scan</td>
<td>Value 24</td>
<td>Informs the table procedure the SQL session is being suspended. The table procedure may release resources in response to this request.</td>
</tr>
</tbody>
</table>
Writing a Table Procedure

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resume Scan</td>
<td>Value 28</td>
<td>Informs the table procedure the indicated scan is being resumed following a suspend. The table procedure may re-establish its state if necessary.</td>
</tr>
</tbody>
</table>

Note: The term scan refers to a set of related operations performed on behalf of one or more SQL statements. A SELECT statement is associated with a separate scan. Similarly, each searched UPDATE or searched DELETE statement is associated with a separate scan. However, all statements referencing the same cursor are associated with the same scan.

Both SELECT statements and OPEN/FETCH/CLOSE cursor requests result in the following set of calls to the table procedure:

- Open Scan
- Next Row (1 to n times)
- Close Scan

A searched UPDATE statement results in the following:

- Open Scan
- Next Row \ (1 to n times)
- Update Row /
- Close Scan

The table procedure is called repeatedly to return the next row to be updated based on the selection criteria passed on the Open Scan request. The results of the Next Row request are examined by the DBMS to determine whether they satisfy all the WHERE clause criteria specified on the searched update statement. If all criteria are satisfied, the table procedure is then called to update the row. If any criteria are not satisfied, the row is not updated and the table procedure is called instead to retrieve the next row.

A positioned UPDATE statement associated with an open cursor has a similar calling sequence except the invoking application determines whether to update the current row.

Searched and positioned DELETE statements result in similar calling sequences to those for searched and positioned UPDATE statements, except a Delete Row request is issued instead of an Update Row request.

INSERT statements result in a single call to the table procedure for each row to be inserted.
Parameter Arguments

On entry to the table procedure, the value of the arguments corresponding to the parameters defined on the CREATE TABLE PROCEDURE statement vary depending on the type of operation performed:

- On an Open Scan request, non-null parameters contain one of the following:
  - Selection criteria specified in the WHERE clause
  - Parameter values specified on the table procedure reference
  - Data type-specific default value if WITH DEFAULT was specified in the table procedure definition

  All other parameters contain nulls (that is, the null indicator for the parameter is negative).

- On an Update Row request, the parameters contain the values returned from the previous Next Row request, overlaid with the values specified in the SET clause of the UPDATE statement.

- On an Insert Row request, the parameters contain the values specified in the VALUES clause of the INSERT statement or the values returned by the SELECT associated with the INSERT statement. Unspecified values are either null or contain the parameter's default value.

- On other types of requests, the contents of the parameters are undefined on entry.

On exit from a Next Row request, the table procedure is expected either to have set the value of the parameter arguments and their indicators appropriately or to have set an SQLSTATE value indicating no-more-rows. If an indicator parameter is set to -1, CA IDMS ignores the value of the corresponding parameter.

Instance Identifier

On every call issued to a table procedure, a parameter is passed identifying the scan to which the request is directed. In the case of INSERT, this has no meaning. However, in all other cases (SELECT, UPDATE, DELETE, and cursor operations) the instance ID can be used to distinguish one scan from another.
Local Work Area

Another parameter passed on each call to a table procedure is a local work area where the table procedure may save information it wishes to preserve from one call to another. Each scan is allocated its own local work area so that values associated with processing an individual scan may be saved appropriately in a local work area. The types of information which you might need to preserve across calls include:

- Subschema control block for a run unit or the session identifier of an SQL session (for retrieval-only table procedures)
- Database position information
- Input parameter values used as selection criteria

CA IDMS allocates a local work area when a scan is opened and frees it when the scan is closed. Each scan receives its own local work area. When the local work area is allocated, it is initialized to binary zeros.

Global Work Area

A global work area is a storage area that can be shared across one or more table procedures or other SQL routines within a transaction. Each global work area has an associated key which is either:

- The four-character identifier specified on the GLOBAL WORK AREA clause
- The fully-qualified name of the table procedure if no identifier was specified

All SQL routines executing within a transaction and having the same global storage key share the same global work area.

Unless transaction sharing is in effect, all SQL routines within an invoking SQL transaction should update the database through only one run unit or SQL transaction to avoid deadlocking. Typically an update table procedure uses a global work area to share the subschema control or SQL session identifier with other SQL routines. A retrieval-only table procedure might instead use only a local work area for each scan, opening the run unit or SQL session on the Open Scan request and terminating it on the Close Scan request.
Chapter 12: Defining and Using Procedures

This section contains the following topics:

When to Use a Procedure (see page 629)
Defining a Procedure (see page 630)
Invoking a Procedure (see page 631)
Procedure Parameters (see page 631)
Writing an External Procedure in COBOL, PL/I or Assembler (see page 634)
Writing External Procedures as CA ADS Mapless Dialogs (see page 636)

When to Use a Procedure

SQL-invoked procedures implement a remote procedure call paradigm. They have similar uses as table procedures, but generally cannot replace table procedures because of their inability to return many rows of a result table with the table procedure parameters as columns of a table. Instead, they return only 0 or 1 row of parameters. Procedures are much simpler to program and can be written in the SQL procedural language.

Procedures can be implemented for many uses which include but is not limited to the following:

- Reuse existing code
- Encapsulate complex code
- Standardize common business processes
- Reduce communication bandwidth
- Easier and faster deployment and control of applications
- Isolate user interface logic from database access
- Implement transparently segmented databases
- Access non-SQL defined databases

While procedures cannot return more than one row from a result table, made up from the procedure parameters, it is possible for the caller of a procedure to receive result sets created by the procedure. The caller can allocate and process dynamic cursors for all the result sets returned by the called procedure. This feature is called dynamic result sets. For more information, see CALL.
Defining a Procedure

How to define and deploy SQL-invoked procedures depends on the language of the procedure.

- For SQL procedures which are specified with LANGUAGE SQL, you define the procedure using the CREATE PROCEDURE statement with the `procedure-statement` clause. After the successful execution and commit of the CREATE PROCEDURE statement, the procedure can be called.

- For external procedures, not specified with LANGUAGE SQL, you must use the following steps:
  1. Define the procedure using the CREATE PROCEDURE statement.
  2. Write the procedure in COBOL, PL/I, Assembler, or CA ADS following the guidelines outlined below. You can also use an existing program as a procedure.
  3. Define the program to a CA IDMS system, if necessary
  4. Invoke the procedure with an SQL CALL statement or from within a query-specification or a SELECT statement.

In the following example, the procedure GET_BONUS is named and associated with schema EMP. The name of the program to be called to service a CALL request of the procedure, CALCSAL, is specified in the EXTERNAL NAME parameter. The PROTOCOL IDMS specifies that the procedure is defined and called using the IDMS protocol. The parameters that pass to and from the procedure are listed. Each parameter definition consists of a name and a data type.

```sql
CREATE PROCEDURE EMP.GET_BONUS
( EMP_ID            UNSIGNED NUMERIC (4),
  START_DATE        DATE,
  SALARY              UNSIGNED NUMERIC (9))
EXTERNAL NAME CALCSAL
PROTOCOL IDMS;
```

More Information

- For more information about syntax and parameters used in defining procedures, see CREATE PROCEDURE.
- For more information and a more detailed example of using CREATE PROCEDURE, see Sample COBOL Procedure, and Sample CA ADS Procedure.
- For more information about procedures returning dynamic result sets, see CALL.
Invoking a Procedure

You invoke procedures using an SQL CALL statement or using a query-specification or a SELECT statement. During SQL CALL processing, CA IDMS issues a call to the corresponding routines. The output parameter values return as a result set.

You can also reference a procedure in the FROM clause of a query-specification or SELECT statement, in the same manner as references to SQL tables, views and table procedures.

If you reference a procedure in a FROM clause, then the parameters of the procedure act as columns in an SQL table or view. You can reference them in SELECT list expressions and WHERE clauses. A procedure returns exactly one row of output or no output. You can reference procedures any place that permits a table reference.

Access to a procedure is controlled in the same way as for a table procedure. GRANT and REVOKE statements on a resource type of TABLE are used to give and remove SELECT or DEFINE privileges on a procedure.

Procedure Parameters

Parameters in procedure references are covered in this section when used with or in the following:

- SQL CALL Statement
- Query specifications and SELECT Statements
- WHERE Clause
Parameters in Procedure References of the SQL CALL Statement

The recommended and easiest way of specifying input parameter values is within the procedure reference itself in the SQL CALL statement. When invoked through the Command Facility, the CALL statement results in a set of output values, one for each parameter defined to the procedure. In embedded SQL a CALL statement results in an output value for each parameter specified as a host variable, a local variable or a routine parameter. For dynamically prepared CALL statements, only the parameters specified in the procedure reference of the CALL will be available as output values.

You specify parameter values supplied on a procedure reference either positionally or as keyword/value pairs. You can also combine them with WHERE clause references to form the set of values that pass to the procedure.

Examples of specifying parameter values

The example below shows different ways in which you can specify input parameter values using the SQL CALL statement.

```sql
CALL EMP.GET_BONUS (EMP_ID = 127, START_YEAR='1998')
CALL EMP.GET_BONUS (127, '1998')
```

Parameters in Procedure References in Query-specifications and SELECT Statements

The parameters associated with a procedure are treated like columns of a table. You can specify them within the column list of a SELECT or a query-specification or the search criteria of a WHERE clause. Additionally, you can specify parameter values within the procedure reference itself.

Column list references

<table>
<thead>
<tr>
<th>Parameters referenced in</th>
<th>Specify</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column list of a SELECT statement</td>
<td>The columns that return to the invoking application</td>
</tr>
</tbody>
</table>
**WHERE Clause References**

You use WHERE clause references to parameters to filter the output of the procedure. Each time a procedure returns a set of output values, they are evaluated against the selection criteria specified in the WHERE clause. A non-conforming "row" results in an SQLSTATE of No Data for the initiating SQL request.

Additionally, you can use WHERE clause parameter references to pass input values to the procedure. If you specify selection criteria in the form of an "=\" comparison (that is, parameter = value), value passes to the procedure. Other types of selection criteria such as IN predicates or comparison predicates with > or < operators have no effect on the value of the parameters passed to the procedure.

Specifically, a reference to a parameter in a WHERE clause results in an input value passing to the procedure only if:

- It appears within an equality test
- The equality test is not combined with other predicates in the WHERE clause through the use of the OR operator
- The NOT operator does not precede the equality test

**WHERE clause Parameter References**

The examples below illustrate how a parameter reference in a WHERE clause affects the value passed to the procedure:

<table>
<thead>
<tr>
<th>WHERE clause</th>
<th>Parameter value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 = 1</td>
<td>1</td>
</tr>
<tr>
<td>P1 &lt; 1</td>
<td>-null-</td>
</tr>
<tr>
<td>P1 = C1</td>
<td>C1</td>
</tr>
<tr>
<td>P1 = 2 AND P2 = 3</td>
<td>2, 3</td>
</tr>
<tr>
<td>P1 = 2 OR P2 &gt; 3</td>
<td>-null-</td>
</tr>
<tr>
<td>P1 IN (2, 3, 8)</td>
<td>-null-</td>
</tr>
</tbody>
</table>

**Difference between procedure reference and WHERE clause**

One difference exists between parameter values specified through a WHERE clause and those specified within the procedure reference. Parameter values specified within the procedure reference are not used to filter the output from the procedure as is the case for those specified within the WHERE clause. Parameter values specified within the procedure reference affect only the input to the procedure and not the output from the procedure. Therefore, the above three select statements are equivalent only if the procedure enforces the conditions specified through the procedure reference.
Writing an External Procedure in COBOL, PL/I or Assembler

You can write the program associated with an external procedure in COBOL, PL/I or Assembler. This requires the procedure to be defined with PROTOCOL IDMS. When called, the program is passed a fixed parameter list consisting of the parameters specified on the procedure definition as well as additional parameters used for communication between CA IDMS and the procedure.

Whenever you make a reference to a procedure, CA IDMS calls the program associated with the procedure to service the request. The procedure responds by processing the input parameters. You can optionally set an error condition in SQLSTATE.

CA IDMS performs transaction and session management automatically in response to requests that the originating application issues. Changes to the database made by a procedure are committed or rolled out together with other changes made within the SQL transaction. The procedure requires no special action to ensure this occurs.

The next section discusses writing a procedure in detail.

For an example of a procedure written in COBOL, see Sample COBOL Procedure.

Calling Arguments

The following sets of arguments pass each time you call a procedure:

- One argument for each of the parameters specified on the procedure definition, passes in the order you declare the parameters
- One argument for each null indicator associated with a parameter specified in the procedure definition, passes in the order you declare the parameters
- A set of common arguments used for communications between CA IDMS and the procedure

The first two sets of arguments vary from one procedure to another. They are used to pass selection criteria and insert/update values to the procedure and result values from the procedure.

The last set of arguments, shown in the table below, is the same for all procedures.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result Indicator (fullword)</td>
<td>Not used</td>
</tr>
</tbody>
</table>
Argument | Contents
---|---
SQLSTATE (CHAR (5)) | Status code returned by the procedure: The initial value is always 00000
  ■ 00000—Indicates success
  ■ 01Hxx—Indicates a warning
  ■ 02000—Indicates no more rows
  ■ 38xxx—Indicates an error
Procedure Name (CHAR (18)) | Name of the procedure
Explicit Name | Not used
Message Text (CHAR (80)) | Message text returned by the procedure and displayed by CA IDMS in the event of an error or warning
SQL Command Code (fullword) | Always 16, indicating a Fetch SQL request.
SQL Operation Code (fullword) | Always 16, indicating a "next row" request.
Instance Identifier (fullword) | Not meaningful for procedures
Local Work Area (User-defined) | A user-defined working storage area
Global Work Area (User-defined) | A user-defined storage area that can be shared by other SQL routines within a transaction.

**Parameter Arguments**

On entry to the procedure, the value of the arguments corresponding to the parameters defined on the CREATE PROCEDURE statement are as follows:

Non-null parameters contain one of the following:

- The parameter values specified on the procedure reference
- The selection criteria specified in the WHERE clause
- The data type-specific default value if WITH DEFAULT was specified in the procedure definition

All other parameters contain nulls (that is, the null indicator for the parameter is negative).

On exit expect the procedure to either have set the value of the parameter arguments and their indicators appropriately or to have set an SQLSTATE value indicating no-more-rows. If you set an indicator parameter to -1, CA IDMS ignores the value of the corresponding parameter.
Local Work Area

Another parameter passed on each call to a procedure is a local work area.

CA IDMS allocates the local work area just before calling the procedure and frees it immediately after the procedure exits. When CA IDMS allocates the local work area, it is initialized to binary zeros.

Global Work Area

A global work area is a storage area that can be shared across one or more procedures or other SQL routines within a transaction. Each global work area has an associated key which is either:

- The four-character identifier specified on the GLOBAL WORK AREA clause
- The fully-qualified name of the procedure if you do not specify an identifier

All SQL routines executing within a transaction and having the same global storage key share the same global work area.

Unless transaction sharing is in effect, all SQL routines within an invoking SQL transaction should update the database through only one run unit or SQL transaction to avoid deadlocking. Typically, an update procedure uses a global work area to share the subschema control or SQL session identifier with other SQL routines. A retrieval-only procedure might instead use only a local work area opening the run unit or SQL session and terminating it on exit.

Writing External Procedures as CA ADS Mapless Dialogs

You can also code an SQL procedure as a CA ADS mapless dialog. This requires the procedure to be defined with PROTOCOL ADS and SYSTEM MODE. The name of the dialog that is loaded and run when the SQL procedure is invoked is specified in the EXTERNAL NAME clause of the CREATE/ALTER PROCEDURE statement.

Note: For more information about coding an SQL procedure as a CA ADS, see CREATE PROCEDURE and the examples given in Sample CA ADS Procedure.

Mapless Dialog

The ADS dialog that implements the SQL procedure must be mapless. The name of this mapless dialog is specified as external-routine-name in the external clause of the procedure definition.

To return to the SQL engine, the CA ADS premap process of the mapless dialog must issue a LEAVE ADS command.
Work Records

To access the procedure parameters, the dialog must include a work record whose name is schema-name.procedure-identifier. This record is not copied from the dictionary but instead is automatically constructed by the CA ADS dialog compiler (ADSC or ADSOBCOM) when it compiles the dialog. You can refer to the procedure parameters in the ADS process code in the same way as you refer to columns in any SQL table. Null indicator(s) (variables) can be referenced by appending "-I" to the relevant column name (see Usage).

When parameters of a procedure are dropped, added or altered, the dialog that implements the procedure must be recompiled. Failure to do so may result in a DC171066 error message when the procedure is next executed. The runtime validation producing this message is based solely on the size of the record.

Additional Records

Besides the pseudo-work record, schema-name.procedure-identifier, other records related to the procedure can be included.

ADSO-SQLPROC-COM-AREA is a system-supplied record. The record layout is given next:

```
ADD RECORD NAME ADSO-SQLPROC-COM-AREA.
    03 FILLER        PIC S9(8) COMP SYNC.
    03 FILLER        PIC X(3).
    03 SQLPROC-SQLSTATE  PIC X(5).
    03 SQLPROC-NAME     PIC X(18).
    03 SQLPROC-SPECIFIC-NAME PIC X(18).
    03 SQLPROC-MSG-TEXT   PIC X(80).
    03 SQLPROC-COMMAND-CODE PIC S9(8) COMP SYNC.
    03 SQLPROC-OPERATION-CODE PIC S9(8) COMP SYNC.
    03 SQLPROC-INSTANCE-ID PIC S9(8) COMP SYNC.
    03 FILLER          OCCURS 2.
```
The non-FILLER elements of the ADSO-SQLPROC-COM-AREA record are the parameters that are common to all SQL procedures. For a description of these parameters, see Calling Arguments.

If the procedure definition contains a LOCAL or GLOBAL WORKAREA clause, you can define corresponding records in the dictionary. While the layout of these records is application dependent, the name must comply with the following rules in order for the ADS runtime to properly initialize these records:

- dialogname-SQLPROC-GLOBAL-AREA
- dialogname-SQLPROC-LOCAL-AREA

The *dialogname* is the name of the dialog, specified as external-routine-name in the external name clause of the procedure definition.

**Note:** For more information and examples, see Sample CA ADS Procedure.
When to Use a User-Defined Function

You can use a user-defined SQL function just as you would use any SQL scalar function. A scalar function is a function whose argument includes zero or more value expressions on which the function operates. The result of a scalar function is a single value. This value is derived from the expression or expressions named in the arguments.
Defining a Function

How to define and deploy user-defined functions depends on the language of the function.

■ For SQL functions, specified with LANGUAGE SQL, define the function using the CREATE FUNCTION statement with the procedure-statement clause. After the successful execution and commit of the create function statement, the function can be invoked.

■ For external functions, not specified with LANGUAGE SQL, complete the following steps:
  1. Define the function using the CREATE FUNCTION statement.
  2. Write the function in COBOL, PL/I, Assembler, or CA ADS following the guidelines given in this chapter. You may also be able to use an existing program as a template for a function.
  3. Define the function to a CA IDMS system, if necessary.
  4. Invoke the function as needed by specifying it anywhere that a value-expression can be specified in an SQL statement.

Note: You invoke the SQL function in a way very similar to the way in which you invoke built-in functions.

An example is shown next:

```
CREATE FUNCTION DEFJE01.UDF_FUNBONUS
  ( EMP_ID      DECIMAL(4) )
RETURNS DECIMAL(10)
EXTERNAL NAME FUNBONUS PROTOCOL IDMS
DEFAULT DATABASE CURRENT
USER MODE
LOCAL WORK AREA 0
;
```

Similarly, use the ALTER FUNCTION and DROP FUNCTION statements to modify and delete the definition of existing functions.

More Information

■ For more information about the syntax and parameters used in defining functions, see CREATE FUNCTION, ALTER FUNCTION, and DROP FUNCTION.

■ For more information and detailed examples about using a CREATE FUNCTION, see Sample COBOL Function, and Sample CA ADS Function.
Invoking a Function

User-defined SQL functions are invoked using the user-defined-function invocation syntax.

Access to user-defined functions is controlled in the same way as for procedures. GRANT and REVOKE statements on a resource type of TABLE are used to give and remove SELECT or DEFINE privileges on a function.

Note: For more information about a user-defined-function, see Expansion of Value-expression and Expansion of User-defined-function.

Writing an External Function in COBOL, PL/I, or Assembler

You can write the program associated with a function in COBOL, PL/I or Assembler. This requires the function to be defined with PROTOCOL IDMS. When called, the program is passed a fixed parameter list consisting of the parameters specified in the function definition, as well as additional parameters used for communication between CA IDMS and the function.

Whenever a function is invoked, CA IDMS calls the program associated with the function to service the request. The function responds by processing the input parameters. An error condition can optionally be set in SQLSTATE.

CA IDMS performs transaction and session management automatically in response to requests that the originating application issues. Changes to the database made by a function are committed or rolled out together with other changes made within the SQL transaction. No special action is required of the function in order to ensure that this occurs.

The next section discusses writing a function in detail.

For an example of a function written in COBOL, see Sample COBOL Function.
Calling Arguments

The following sets of arguments are passed when a function is called:

- One argument for each of the parameters specified on the function definition, passed in the order in which the parameters were declared. These arguments vary from function to function; they are used to pass values to the function.
- One argument to contain the return value of the function. The implicit name for this argument is USER_FUNC.
- One argument for each null indicator associated with a parameter specified in the function definition, passed in the order in which the parameters were declared. These arguments vary from function to function; they are used to pass values to the function.
- One argument for the null indicator associated with the return value of the function (the null indicator for the USER_FUNC parameter).
- A set of common arguments used for communications between CA IDMS and the function. This set of arguments, shown in the following table, is the same for all functions.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result Indicator (fullword)</td>
<td>Not used</td>
</tr>
<tr>
<td>SQLSTATE (CHAR (5))</td>
<td>Status code returned by the procedure:</td>
</tr>
<tr>
<td></td>
<td>The initial value is always 00000</td>
</tr>
<tr>
<td></td>
<td>00000 Indicates success</td>
</tr>
<tr>
<td></td>
<td>01Hxx Indicates a warning</td>
</tr>
<tr>
<td></td>
<td>02000 Indicates no more rows</td>
</tr>
<tr>
<td></td>
<td>38xxx Indicates an error</td>
</tr>
<tr>
<td>Function Name (CHAR (18))</td>
<td>Name of the function</td>
</tr>
<tr>
<td>Explicit Name</td>
<td>Not used</td>
</tr>
<tr>
<td>Message Text (CHAR (80))</td>
<td>Message text returned by the function and displayed by CA IDMS in the event of an error or warning</td>
</tr>
<tr>
<td>SQL Command Code (fullword)</td>
<td>Always 16, indicating a Fetch SQL request</td>
</tr>
<tr>
<td>SQL Operation Code (fullword)</td>
<td>Always 16, indicating a &quot;next row&quot; request</td>
</tr>
<tr>
<td>Instance Identifier (fullword)</td>
<td>Not meaningful for functions</td>
</tr>
<tr>
<td>Local Work Area (user-defined)</td>
<td>A user-defined working storage area</td>
</tr>
<tr>
<td>Global Work Area (user-defined)</td>
<td>A user-defined storage area that can be shared by other SQL routines within a transaction.</td>
</tr>
</tbody>
</table>
Parameter Arguments

On entry to the program associated with a function, the value of the arguments corresponding to the parameters defined in the CREATE FUNCTION statement are as follows:

- Non-null parameters contain one of the following:
  - The parameter values specified on the function reference
  - The data type-specific default value if WITH DEFAULT was specified in the function definition, and no value was specified in the function invocation
- All other parameters contain nulls (that is, the null indicator for the parameter is negative).

On exit, the program associated with the function is expected either to have set the value of the parameter USER_FUNC, holding the functions return value and the corresponding indicator appropriately, or to have set an SQLSTATE value indicating no-more-rows. If the indicator parameter is set to -1, CA IDMS ignores the value of the USER_FUNC parameter.

Local Work Area

Another parameter passed on each call to a function is a local work area.

CA IDMS allocates the local work area just before calling the function and frees it immediately after the function exits. When the local work area is allocated, it is initialized to binary zeros.

Global Work Area

A global work area is a storage area that can be shared across one or more functions or other SQL routines within a transaction. Each global work area has an associated key that is one of the following:

- Four-character identifier specified on the GLOBAL WORK AREA clause
- Fully-qualified name of the function if no identifier was specified

All SQL routines executing within a transaction and having the same global storage key share the same global work area.
Writing External Functions as CA ADS Mapless Dialogs

You can also code an SQL function as a CA ADS mapless dialog. This requires the function to be defined with PROTOCOL ADS and SYSTEM MODE. The name of the dialog that is loaded and run when the SQL function is invoked is specified in the EXTERNAL NAME clause of the CREATE/ALTER FUNCTION statement.

Note: For more information, see CREATE FUNCTION and the examples given in Sample CA ADS Function.

Mapless Dialog

The ADS dialog that implements the SQL function must be mapless. The name of this mapless dialog is specified as external-routine-name in the external name clause of the function definition.

To return to the SQL engine, the CA ADS premap process of the mapless dialog must issue a LEAVE ADS.

Work Records

To access the function parameters and the result parameter USER_FUNC, the dialog must include a work record whose name is schema-name.function-identifier. This record is not copied from the dictionary but instead is automatically constructed by the CA ADS dialog compiler (ADSC or ADSOBCOM) when it compiles the dialog. You can refer to the function parameters and the result parameter USER_FUNC and the corresponding null indicators in the ADS process code in the same way as you refer to columns in any SQL table.

When parameters of a function are dropped, added or altered, the dialog that implements the SQL function must be recompiled. Failure to do so may result in a DC171066 error message when the function is next executed. The runtime validation producing this message is based solely on the size of the record.
Additional Records

Besides the pseudo-work-record, schema-name.function-identifier, other records related to the function can be included.

ADSO-SQLPROC-COM-AREA is a system-supplied record. The record layout is shown next:

```
AD RECORD NAME ADSO-SQLPROC-COM-AREA.
  03 FILLER PIC S9(8) COMP SYNC.
  03 FILLER PIC X(3).
  03 SQLPROC-SQLSTATE PIC X(5).
  03 SQLPROC-NAME PIC X(18).
  03 SQLPROC-SPECIFIC-NAME PIC X(18).
  03 SQLPROC-MSG-TEXT PIC X(80).
  03 SQLPROC-COMMAND-CODE PIC S9(8) COMP SYNC.
  03 SQLPROC-OPERATION-CODE PIC S9(8) COMP SYNC.
  03 SQLPROC-INSTANCE-ID PIC S9(8) COMP SYNC.
  03 FILLER OCCURS 2.
```

The non-FILLER elements of the ADSO-SQLPROC-COM-AREA record are the parameters that are common to all SQL functions. For a description of these parameters, see Calling Arguments.

If the function definition contains a LOCAL or GLOBAL WORKAREA clause, you can define corresponding records in the dictionary. While the layout of these records is application dependent, the name must comply with the following rules for the ADS runtime to properly initialize these records:

- dialogname-SQLPROC-GLOBAL-AREA
- dialogname-SQLPROC-LOCAL-AREA

The `dialogname` is the name of the dialog, specified as external-routine-name in the external name clause of the function definition.
Chapter 14: Considerations for SQL-invoked External Routines

This section contains the following topics:

- Special Considerations for SQL-invoked External Routines (see page 647)
- Debugging Procedures (see page 653)
- Database Name Inheritance (see page 653)
- Transaction Sharing (see page 654)

Special Considerations for SQL-invoked External Routines

This section discusses the following special considerations for table procedures, procedures and functions, commonly referred to as SQL-invoked external routines.

- Environment independence
- Transaction management
- Suspension/resumption of SQL-invoked external routines
- Error handling
- Datetime parameters
- Transaction mode
**Environment Independence**

Since it is likely an SQL-invoked external routine will execute within a batch address space because of local mode access and within the DC/UCF address space, it should be independent of the runtime environment.

If your SQL-invoked external routine issues only database requests, use a protocol mode of BATCH to ensure it executes in local mode and within the DC/UCF address space.

If an SQL-invoked external routine is executed in local mode (for example, through IDMSBCF), it must limit itself to database requests only or be written in Assembler (or use an Assembler subroutine for DC/UCF requests) or CA ADS.

Even if the SQL-invoked external routine is written in Assembler, many DC services such as Print, Queue and Terminal I/O are not supported in local mode and should not be used. Scratch and storage requests issued from an assembler program are supported in local mode. Terminal I/O services are not supported in local mode or within the DC/UCF address space.

If an SQL-invoked external routine is executed within DC/UCF, it should not contain statements that interfere with, or are prohibited from, that environment. For example, you should avoid DISPLAY statements in COBOL and GETMAIN requests in Assembler. Follow the rules specified in the appropriate *CA IDMS DML Reference Guide* when coding your SQL-invoked external routine.

When an SQL-invoked external routine is written as a mapless dialog in CA ADS, the CA ADS batch environment must have been configured resulting in an ADSOOPTI load module being available in the local mode load libraries.
Transaction Management

Run units and SQL transactions opened within an SQL-invoked external routine are managed automatically as part of the invoking SQL session’s transaction. If the invoking application issues a COMMIT WORK, all subordinate transactions opened by SQL-invoked external routines are also committed. Similarly, if the invoking SQL session is rolled out, all subordinate transactions are also rolled out.

Although an SQL-invoked external routine is free to terminate its own transactions independently from the invoking SQL session, it should do so only if it made no changes to the database.

Terminating the invoking SQL transaction, either through a COMMIT or ROLLBACK operation, affects SQL-invoked external routines in the following ways:

- All open scans are closed. A Close Scan request is issued to each affected SQL-invoked external routine.
- All database transactions (SQL or non-SQL) started by the SQL-invoked external routine are either committed or rolled out and the corresponding sessions are terminated.
- All SQL-invoked external routine work areas are freed.

When the invoking SQL transaction is committed through a COMMIT CONTINUE operation, the only effect on SQL-invoked external routines is that database changes that the SQL-invoked external routine made are committed.
Special Considerations for SQL-invoked External Routines

Suspend/Resume

If an SQL session which invokes an SQL-invoked external routine is suspended, the SQL-invoked external routine is also suspended, with the following results:

- Run units and SQL sessions started by the SQL-invoked external routine are suspended
- Database changes made by the SQL-invoked external routine (whether through SQL or native DML) are neither committed nor rolled out; instead the records remain locked and the changes are either committed or rolled out when the invoking SQL session is committed or rolled out
- Work areas associated with the SQL-invoked external routine are retained

In most cases, no special action is required of the SQL-invoked external routine during a suspend operation. The SQL-invoked external routine is called once for each open scan that exists at the time the session is suspended. If the SQL-invoked external routine has acquired temporary storage, it might need to save its contents somewhere else (such as in kept storage or in a scratch area) that is preserved across a pseudo-converse; otherwise, the SQL-invoked external routine may ignore Suspend Scan requests.

Similarly, the SQL-invoked external routine may ignore Resume Scan requests unless it needs to restore a temporary storage area. The first request to an SQL-invoked external routine after a resume is a Resume Scan, if the request involves a scan that was previously suspended. If the request does not involve a previously-suspended scan, the SQL-invoked external routine might not be aware that a suspend and resume has occurred. The contents of the SQL-invoked external routine’s work areas are the same as before the suspend, and any run units or SQL sessions previously started by the SQL-invoked external routine resume automatically on the next database request.

Error Handling

The SQL-invoked external routine has two arguments to signal an exception condition back to CA IDMS. These arguments consist of a five-character SQLSTATE code and an 80-byte message area. The following table lists valid SQLSTATE codes and their descriptions.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>Request was successful</td>
</tr>
<tr>
<td>01Hxx</td>
<td>Request was successful but the SQL-invoked external routine generated a warning message</td>
</tr>
<tr>
<td>02000</td>
<td>No more rows to be returned</td>
</tr>
<tr>
<td>38xxx</td>
<td>The SQL-invoked external routine has detected an error during processing</td>
</tr>
</tbody>
</table>
CA IDMS examines the SQLSTATE value to determine whether the operation was successful. An SQLSTATE value of 0200 indicates that all rows have been returned. It is meaningful only on a Next Row request and cannot be set while at the same time returning a row since CA IDMS ignores parameter arguments if an SQLSTATE value of 02000 is set.

If an SQLSTATE value indicates that an error or warning condition exists, CA IDMS embeds the message text returned by the SQL-invoked external routine in a standard DB message and returns it to the calling application through the message area of the SQLCA. It also translates the SQLSTATE value into one of the following SQLCODE values:

<table>
<thead>
<tr>
<th>SQLSTATE Value</th>
<th>SQLCODE Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>0</td>
</tr>
<tr>
<td>01Hxx</td>
<td>1</td>
</tr>
<tr>
<td>02000</td>
<td>100</td>
</tr>
<tr>
<td>38xxx</td>
<td>-4</td>
</tr>
</tbody>
</table>

If the SQL-invoked external routine signals an error, CA IDMS automatically rolls out all database changes that the SQL-invoked external routine made while processing the SQL statement that caused the SQL-invoked external routine to be invoked. For example, if the invoking SQL statement was a searched update and ten rows had been updated before the error was detected, changes to all ten rows are rolled out automatically. Database changes made prior to the execution of the searched update statement are not rolled out.

**Datetime Parameters**

If an SQL-invoked external routine has a parameter with a data type of DATE, TIME, or TIMESTAMP, the values passed to and from the SQL-invoked external routine are in the eight-byte internal datetime format. To interpret incoming parameters, the SQL-invoked external routine must first convert them to external format using either the IDMSIN01 subroutine or the #XTRA macro. Similarly, before returning a datetime parameter value, the SQL-invoked external routine must convert the external format to the internal format again using either IDMSIN01 or #XTRA.

To avoid this type of conversion, you can define date/time parameters using a character data type which may then be converted to a DATE, TIME, or TIMESTAMP in DML statements using the CAST function. However, this method requires that the invoking application or end-user specify the CAST operation, and that the SQL-invoked external routine validate the datetime on update and insert values.

**Note:** For more information about datetime parameters, see "Calls to IDMSIN01" in the CA IDMS Callable Services Guide.
Special Considerations for SQL-invoked External Routines

Transaction Mode

The transaction mode of an SQL session which invokes an SQL-invoked external routine is propagated to the subordinate transactions started by the SQL-invoked external routine. If the SQL-invoked external routine starts an SQL session, its default transaction mode is the same as the transaction mode associated with the invoking transaction. If the SQL-invoked external routine binds a run unit and the transaction mode of the invoking transaction is READ ONLY, all update ready modes are converted automatically to SHARED RETRIEVAL. The net result is that an SQL-invoked external routine invoked by a transaction in a READ ONLY state is unable to update the database.

DC/UCF Program Definition

If the SQL-invoked external routine is executed within the DC/UCF address space, you must define the program to the DC/UCF system using one of the following:

- DCMT VARY DYNAMIC PROGRAM command
- ADD PROGRAM system generation statement

Compile and Link Options

Compile and link options vary depending on the language in which the SQL-invoked external routine is written and, in some cases, on the version of the compiler used. The following guidelines apply to compile and link options for SQL-invoked external routines:

- Programs should be reentrant or pseudo-reentrant.
- Programs should be linked with an AMODE of 31. If you do not do this, you must define all invoking tasks, including OCF, as LOC=BELLOW.

COBOL Working Storage

You should not rely on the contents of COBOL working storage to be retained across calls, nor should you rely on initialization to establish values on the first request for a scan if the value might have been altered in a prior request.

The allocation and initialization of working storage varies depending on whether:

- The SQL-invoked external routine is invoked in local mode or within the DC/UCF address space
- Concurrent scans are being processed

Use working storage only for the following:

- Constants the SQL-invoked external routine will never change
- Variables that are initialized and used within a single call to the program

Other types of data, such as first-time flags or variables set in one call that are used within another, must be defined within either a local or global work area.
Debugging Procedures

The following techniques can help you debug your SQL-invoked external routines:

- Use WRITE TO LOG or SNAP requests to display trace information and key data structures to the log (available under DC/UCF only).
- Use COBOL or PL/I DISPLAY statements or write messages to a print file (available under local mode or with LE/370 support under DC)
  
  **Note:** If the program uses a VS COBOL compiler, even in local mode, you cannot use:
  - DISPLAY statements
  - STATE and FLOW compiler options
- Use SQLTRACE and DMLTRACE SYSIDMS parameters to trace the database calls made by the SQL-invoked external routine (available under local mode only).
- Use PROCTRACE in conjunction with SQLTRACE or DMLTRACE to trace the calls made into and out of the SQL-invoked external routine. This option displays the parameters both before and after the call (available under local mode only)

  **Note:** You enable SQL-invoked external routine tracing by including the following parameters in the SYSIDMS file:

  SQLTRACE=ON or DMLTRACE=ON
  PROCTRACE=ON

Database Name Inheritance

An SQL-invoked external routine can inherit the current database of the current session. The DEFAULT DATABASE attribute of the SQL-invoked external routine definition controls the inheritance of the default database as follows:

- DEFAULT DATABASE NULL guarantees compatibility with previous releases of CA IDMS.
- DEFAULT DATABASE CURRENT makes the current database the default database for any subordinate database session started by the SQL-invoked external routine.
Transaction Sharing

Transaction sharing allows multiple database sessions within a user session to share a single locking structure and recovery unit, thereby eliminating inter-session deadlocks.

Any access to a database from within an SQL-invoked external routine brings with it the potential for deadlocking if the same data is directly accessed from within the encompassing SQL session. By having both the SQL-invoked external routines and the encompassing SQL session all share a single transaction, the deadlock potential is eliminated.

Transaction sharing for SQL-invoked external routines is controlled by the TRANSACTION SHARING attribute of the SQL-invoked external routine definition. See the different SQL DDL statements for SQL-invoked external routines.
Chapter 15: XML Publishing Using SQL

This section contains the following topics:

- XML Publishing (see page 655)
- XMLSLICE Table Procedure (see page 663)

XML Publishing

XML Publishing allows applications to generate XML data from data stored in a CA IDMS database easily and with high performance. Although the API is based on SQL, CA IDMS SQL supports SQL DML on non-SQL defined databases. Thus, also allowing non-SQL defined CA IDMS databases to be used as the data sources for XML Publishing.

XML Publishing is based on and implements a subset of the SQL/XML ISO standard as described in the ISO publication WD ISO/IEC 9075-14:2007 (E), titled "Information technology - Database languages - SQL - Part 14: XML-Related Specifications (SQL/XML)". A few extensions have been made available. These are indicated in the following section.

The XML Publishing capability is made available through a set of SQL functions, a new internal XML data type, an SQL table procedure, and an XML encoding session option.
SQL/XML Functions

The SQL/XML functions are not true SQL functions but pseudo functions. Some of the SQL/XML functions:

- have a variable number of arguments
- use "AS" to specify an alias for an expression as an argument
- have arguments that can be of any type
- support subqueries as arguments
- have arguments that can be SQL identifiers

Following is a list of the SQL/XML routines (all functions except for one table procedure) that can be used for XML Publishing purposes:

XML Value Functions

- **XMLAGG**—aggregates a set of XML values.
- **XMLATTRIBUTES**—function-like construct, only allowed as argument of XMLELEMENT. Generates XML attributes.
- **XMLCOMMENT**—generates an XML comment.
- **XMLCONCAT**—concatenates multiple XML values.
- **XMLELEMENT**—generates an XML element.
- **XMLFOREST**—generates a forest (collection) of XML elements.
- **XMLNAMESPACES**—function-like construct, only allowed as argument of XMLELEMENT or XMLFOREST. Generates XML namespaces.
- **XMLPARSE**—checks if an XML value is well-formed.
- **XMLPI**—generates an XML processing instruction, that is, an XML declaration or style sheet specification.
- **XMLROOT**—sets the XML version and standalone option in the XML declaration of a root XML element. If an XML declaration is not yet present, one is created with the ENCODING pseudo attribute set to the session's current XML encoding.

CA IDMS Scalar Functions

- **XMLPOINTER**—returns a pointer to a character large object or CLOB, representing a serialized XML value. This CA IDMS extension can be used in programs, running in the same address space as CA IDMS.
- **XMLSERIALIZE**—returns a character or binary value with a maximum length of 30,000, representing a serialized XML value.

Table Procedure

- **XMLSLICE**—retrieves character or binary slices of equal length from the serialization of an XML value.
More Information

- For more information about detailed syntax and semantics of the SQL/XML routines, see XML Value Functions.
- For more information about detailed syntax and semantics of the XMLPOINTER and XMLSERIALIZE scalar functions, see CA IDMS Scalar Functions.
- For more information about detailed syntax and semantics of the XMLSLICE table procedure, see XMLSLICE Table Procedure.

XML Data Type and XML Values

The XML data type is an internal only data type that represents XML data. XML values are usually used as arguments to some of the SQL/XML functions.

The only way to produce an XML value is through the invocation of an XML value function, possibly indirectly through using a subquery that returns an XML value. The return value of all XML value functions is of the XML data type. A subquery used as an XML-value-expression must be of the XML data type, which implies that its SELECT list contains an XML value function.

Data of the XML type cannot be stored in a database or directly used in application programs using the standard SQL API. Programs running in the CA IDMS CV address space or in batch local mode can access serialized XML data using the XMLPOINTER function. After serialization and casting to CHAR or VARCHAR through the XMLSERIALIZE function, XML data can be accessed using any supported SQL API on any platform.

To bypass the 30,000 length limit of the character string returned by XMLSERIALIZE, use the XMLSLICE table procedure.

Examples of valid XML values are as follows:

- XML element
- Forest of XML elements
- Textual content of an XML element
- NULL value
- XML subquery (Select XMLAGG(XMLELEMENT(…)) from ATABLE where …)

Syntax

No syntax is available. This is a special, internal only data type.
Mappings

SQL and XML are two different languages with their own specific language elements and grammar. When using SQL to produce XML, SQL language elements must be mapped to XML using appropriate rules.

This section describes the rules that are used for mapping of:
- Plain text SQL to XML
- SQL identifier to XML
- SQL data type values to XML schema data type values

Mapping Plain Text SQL to XML

This mapping is between the character set(s) of the SQL language and Unicode.

This feature supports mapping from the SQL character set (EBCDIC) to Unicode using encodings UTF-8, UTF-16-BE (big endian), and UTF-16-LE (little endian). This mapping is implemented using the standard CA IDMS code tables (RHDCCODE) and is controlled through the XML encoding session option. The default is not to map; the serialized XML values are encoded in EBCDIC.

Note: For more information about encoding XML values, see the XML encoding session option under SET SESSION.

Mapping SQL Identifier to XML

You can use a much greater range of characters in an SQL identifier than in an XML name. Any character can be used in an SQL identifier delimited by double quotes.

The normative definition of valid XML Name characters is found in the SQL/XML ISO standard. Valid first characters of XML Names are:
- Letters, <underscore>, and <colon>

Valid XML Name characters, after the first character, are:
- Letters, Digits, <period>, <minus sign>, <underscore>, <colon>, CombiningChars, and Extenders

Note: The XML definition of Letter and Digit is broader than <simple Latin letter> and <digit> respectively.
There are two types of XML names: XML NCName and XML QName. An XML NCName is an XML non-colonized name and contains no colon (:) character. An XML QName is an XML-qualified name that consists of the XML namespace prefix and the local part of the name, separated by a colon (:) character. The namespace prefix and the local part of the name must be XML NCNames. For example, xsd:string is an XML QName, where xsd is the namespace prefix which must have been declared for a namespace URI, and string is the local part of the name.

There are two types of mapping of SQL identifiers to XML: fully escaped and partially escaped. Fully escaped mapping is used for all SQL identifiers that are derived from an SQL column name, such as in the XMLATTRIBUTES and XMLFOREST functions. Partially escaped mapping is used in all the other cases, such as in the AS clause of the XMLATTRIBUTES, XMLFOREST, and XMLNAMESPACES functions, and in the NAME clause of the XMLELEMENT and XMLPI functions.

XML names that begin with the characters "xml" (in any combination) are reserved by W3C for use in future recommendations and cannot be used.

The following table shows some mapping examples:

<table>
<thead>
<tr>
<th>SQL Identifier</th>
<th>Fully Escaped XML Name</th>
<th>Partially Escaped XML Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>department</td>
<td>DEPARTMENT</td>
<td>DEPARTMENT</td>
</tr>
<tr>
<td>&quot;department&quot;</td>
<td>department</td>
<td>department</td>
</tr>
<tr>
<td>&quot;last name&quot;</td>
<td>last_x0020_name</td>
<td>last_x0020_name</td>
</tr>
<tr>
<td>&quot;last_xname&quot;</td>
<td>last_x005f_name</td>
<td>last_x005f_name</td>
</tr>
<tr>
<td>&quot;dept:id&quot;</td>
<td>dept_x003A_id</td>
<td>dept:id</td>
</tr>
<tr>
<td>&quot;:id&quot;</td>
<td>_x003A_id</td>
<td>_x003A_id</td>
</tr>
<tr>
<td>&quot;xmlcolumn&quot;</td>
<td>_x0078_mlcolum</td>
<td>xmlcolumn</td>
</tr>
<tr>
<td>&quot;Xmlcolumn&quot;</td>
<td>_x0058_mlcolum</td>
<td>Xmlcolumn</td>
</tr>
<tr>
<td>xmlcolumn</td>
<td>_x0058_MLCOLUMN</td>
<td>XMLCOLUMN</td>
</tr>
</tbody>
</table>
Mapping SQL Data Type Values to XML Schema Data Type Values

This feature supports mapping SQL data type values to XML schema data type values; however, mapping of GRAPHIC and VARGRAPHIC are not supported.

You can map null values using absence or xsi:nil="true".

The complete mapping rules are described in the SQL/XML ISO standard specification. As an oversimplification, mapping can be described as the result of the casting of the SQL data value to VARCHAR(max).

The following table shows some of the character value mappings:

<table>
<thead>
<tr>
<th>SQL Character Value</th>
<th>Mapped Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>&amp;lt .</td>
</tr>
<tr>
<td>&gt;</td>
<td>&amp;gt .</td>
</tr>
<tr>
<td>&amp;</td>
<td>&amp;amp .</td>
</tr>
<tr>
<td>Carriage Return</td>
<td>&amp;#x0d .</td>
</tr>
<tr>
<td>'</td>
<td>&amp;apos .</td>
</tr>
<tr>
<td>&quot;</td>
<td>&amp;quot .</td>
</tr>
</tbody>
</table>

This mapping does not apply to the characters belonging to an XML CDATA section; a CDATA section begins with the string "<![CDATA[" and ends with the string "]>".

Example

In the following example, the use of many of the SQL/XML functions is shown. The result of the SELECT is an XML document that contains all the employees from the DEMOEMPL.EMPLOYEE table, grouped by department. The DEMOEMPL.EMPLOYEE and DEMOEMPL.DEPARTMENT tables are equi-joined on the DEPT_ID. To limit the size of the output, a WHERE clause is coded for the DEPT_ID column. Note how the SELECT statement clearly and naturally reflects the structure of the XML document.
select
XMLSERIALIZE(CONTENT
XMLCONCAT(
XMLPI(NAME "xml"
,'version="1.0" encoding="UTF-8" standalone="yes"")
,XMLELEMENT(NAME "EmployeesByDepartment"
,XMLAGG
XMLELEMENT(NAME "Department"
,XMLATTRIBUTES(DEPT_ID as "DeptId"
,DEPT_NAME as "DeptName")
,select XMLAGG(
XMLELEMENT(NAME "Employee"
,XMLATTRIBUTES(EMP_ID as "EmpId")
,e.EMP_FNAME
,e.EMP_LNAME
,XMLELEMENT(Name "Address"
,XMLFOREST(
    e.STREET as "Street"
    ,e.CITY as "City"
    ,e.STATE as "State"

),
)
)
)
)
from DEMOEMPL.EMPLOYEE e
where d.DEPT_ID = e.DEPT_ID
)
)
)
)
)
)
)
)
as VARCHAR(5000)
)
from DEMOEMPL.DEPARTMENT d
where d.DEPT_ID < 1120
The result, which has been formatted for clarity, is similar to the following:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<EmployeesByDepartment>
  <Department DeptId="1100" DeptName="PURCHASING - USED CARS">
    <Employee EmpId="5008">
      Timothy Fordman
      <Address>
        <Street>60 Boston Rd</Street>
        <City>Brookline</City>
        <State>MA</State>
      </Address>
    </Employee>
    <Employee EmpId="4703">
      Martin Halloran
      <Address>
        <Street>27 Elm St</Street>
        <City>Brookline</City>
        <State>MA</State>
      </Address>
    </Employee>
    <Employee EmpId="2246">
      Marylou Hamel
      <Address>
        <Street>11 Main St</Street>
        <City>Medford</City>
        <State>MA</State>
      </Address>
    </Employee>
  </Department>
  <Department DeptId="1110" DeptName="PURCHASING - NEW CARS">
    <Employee EmpId="2106">
      Susan Widman
      <Address>
        <Street>43 Oak St</Street>
        <City>Medford</City>
        <State>MA</State>
      </Address>
    </Employee>
    <Employee EmpId="1765">
      David Alexander
      <Address>
        <Street>18 Cross St</Street>
        <City>Grover</City>
        <State>MA</State>
      </Address>
    </Employee>
  </Department>
</EmployeesByDepartment>
```
XMLSLICE Table Procedure

SYSCA.XMLSLICE is a table procedure used to retrieve character or binary slices of equal length from the serialization of an XML value. It is a CA IDMS extension that has been made available to allow any client program to process large serialized XML values. It is used when neither XMLSERIALIZE (limited to 30,000 characters) nor XMLPOINTER (requires client to run in same address space as CA IDMS) can be used.

Syntax

```
SELECT
  CAST
    (SLICE
      AS BIN(slic-size)
    )
    AS CHAR(slic-size)
  
FROM SYSCA.XMLSLICE

WHERE
  SLICESIZE = slice-size
  AND PADDING = X'pad-hex'

XMLVALUE = XML-value-expression
```
**Parameters**

**TOTLENGTH**

Specifies the total length of the serialized XML value, without padding characters.

**RESTLENGTH**

Specifies the XML data length, without padding characters, that have not been returned yet.

**SYS:XMLSLICE**

Specifies a table procedure that slices **XML-value-expression**, after serialization, into slices of equal size, specified by **slice-size**. Each row returned by **SYSCA.XMLSLICE** represents a slice.

**slice-size**

Specifies an integer **value-expression**, with a positive value <= 8192. A **slice-size** must always be present, either as the first positional parameter of the table procedure or as the right operand in the equal predicate for **SLICESIZE**.

**pad-hex**

Specifies an optional two-byte hexadecimal literal that is used to pad the last slice of the serialized XML value. The default depends on the XML ENCODING parameter of the SQL SET SESSION statement. Two spaces are used for EBCDIC (X'4040') and UTF8 (X'2020'); one space is used for UTF16LE (X'2000') and UTF16BE (X'0020').

If **XML-value-expression** is specified as a **subquery**, it must be enclosed in parentheses.

The content of the slice is available in the SLICE column of the table procedure. Optionally, you can specify additional columns in the SELECT statement.
Examples

Example 1

In the following SELECT statement, all the employees in DEMOEMP.EMPLOYEE are aggregated in one XML value. This XML value is serialized, and each row returned is a 40-character slice of the serialized XML value.

```sql
select cast(slice as char(40)) from SYSCA.XMLSLICE
where slicesize = 40
  and xmlvalue =
  (select xmlagg(xmlelement(name "employee",
                                xmlagg(xmlelement(name "Name",
                                                  E.EMP_LNAME)))
       from DEMOEMPL.EMPLOYEE e)
)
```

The result is similar to the following:

```xml
<employee><Name>Albertini</Name><Name>Albertini
<Name>Albertini</Name>....
<Name>Alexander</Name><Name>Alexander
<Name>Anderson</Name><Name>Anderson
<Name>Baldwin</Name><Name>Baldwin
<Name>Bennett</Name><Name>Bennett
<Name>Bradley</Name><Name>Bradley
<Name>Brooks</Name><Name>Brooks
<Name>Carlson</Name><Name>Carlson
<Name>Catlin</Name><Name>Catlin
<Name>Courteney</Name><Name>Courteney
<Name>Crane</Name><Name>Crane
<Name>Cromwell</Name><Name>Cromwell
<Name>Dexter</Name><Name>Dexter
<Name>Dixon</Name><Name>Dixon
<Name>Ferguson</Name><Name>Ferguson
<Name>Ferndale</Name><Name>Ferndale
<Name>Fordman</Name><Name>Fordman
<Name>Gallway</Name><Name>Gallway
<Name>Griffin</Name><Name>Griffin
<Name>Hall</Name><Name>Hall
```
Example 2

This example shows the z/OS JCL for the batch command facility IDMSBCF and the SQL statements to create the z/OS dataset "CAIDMS.SAMPLE.XML" holding an XML document encoded in Unicode UTF-16 Little Endian.

The XML document contains the id and name of all employees as present in the DEMOEMPL.EMPLOYEE table.

With a binary file transfer, the dataset can be copied to other platforms for further processing. The use of the XMLSLICE table procedure allows for creating XML documents up to 2 GB.

```
//BCFLOCAL EXEC PGM=IDMSBCF,REGION=7500K
//STEPLIB DD DSN=CAIDMS.R160.LOADLIB,DISP=SHR
//SYSLST DD SYSOUT=A
//SYSOUT DD SYSOUT=A
//OUTPUT DD DSN=CAIDMS.SAMPLE.XML,DISP=(NEW,CATLG),
//     DCB=(RECFM=FB,LRECL=80,BLKSIZE=16000),
//     UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSIPT DD *
set options OUTPUT to OUTPUT; -- redirects output of BCF
set session XML Encoding utf16LE;-- requests UTF-16 LE encoding
select cast(slice as char(80))
    from sysca.xmlslice(80, X'2000')-- defines slices of 80 bytes
    -- padding with space in UTF-16 LE
    where xmlvalue =
        (select xmlroot(xmlelement(name "AllEmployees"
            , xmlagg(xmlelement(name "Emp"
                , xmlattributes(EMP_ID as "Id")
                , EMP_FNAME ||EMP_LNAME)))
            , version '1.0')
        from DEMOEMPL.EMPLOYEE
    )
```
## Appendix A: Summary Comparison to SQL Standard

### SQL Standard Basis

CA IDMS SQL is based on the ISO/IEC SQL Standards endorsed by ANSI.

### Additional Statements in CA IDMS

CA IDMS supports the following SQL statements not included in the SQL standard:

<table>
<thead>
<tr>
<th>Statement category</th>
<th>CA IDMS extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access module management</td>
<td>ALTER ACCESS MODULE&lt;br&gt;CREATE ACCESS MODULE&lt;br&gt;DROP ACCESS MODULE&lt;br&gt;EXPLAIN</td>
</tr>
<tr>
<td>Authorization</td>
<td>GRANT definition privileges&lt;br&gt;GRANT EXECUTE&lt;br&gt;REVOKE SQL definition privileges&lt;br&gt;REVOKE EXECUTE&lt;br&gt;TRANSFER OWNERSHIP</td>
</tr>
<tr>
<td>Data manipulation</td>
<td>DECLARE EXTERNAL CURSOR</td>
</tr>
</tbody>
</table>
### SQL Standard Basis

#### Statement category

<table>
<thead>
<tr>
<th>Logical data description</th>
<th>CA IDMS extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER CATALOG</td>
<td></td>
</tr>
<tr>
<td>ALTER INDEX</td>
<td></td>
</tr>
<tr>
<td>ALTER SCHEMA</td>
<td></td>
</tr>
<tr>
<td>ALTER TABLE PROCEDURE</td>
<td></td>
</tr>
<tr>
<td>CREATE CALC</td>
<td></td>
</tr>
<tr>
<td>CREATE CONSTRAINT</td>
<td></td>
</tr>
<tr>
<td>CREATE INDEX</td>
<td></td>
</tr>
<tr>
<td>CREATE KEY</td>
<td></td>
</tr>
<tr>
<td>CREATE TABLE PROCEDURE</td>
<td></td>
</tr>
<tr>
<td>CREATE TEMPORARY TABLE</td>
<td></td>
</tr>
<tr>
<td>DROP CALC</td>
<td></td>
</tr>
<tr>
<td>DROP CONSTRAINT</td>
<td></td>
</tr>
<tr>
<td>DROP INDEX</td>
<td></td>
</tr>
<tr>
<td>DROP KEY</td>
<td></td>
</tr>
<tr>
<td>DROP TABLE PROCEDURE</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Precompiler-directive</th>
<th>INCLUDE</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Session management</th>
<th>RELEASE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RESUME SESSION</td>
</tr>
<tr>
<td></td>
<td>SET SESSION</td>
</tr>
<tr>
<td></td>
<td>SUSPEND SESSION</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transaction management</th>
<th>SET ACCESS MODULE</th>
</tr>
</thead>
</table>

### Additional Parameters and Capabilities in CA IDMS

CA IDMS supports the following additional parameters and capabilities not included in the SQL standard:

<table>
<thead>
<tr>
<th>Statement or component</th>
<th>CA IDMS extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifiers</td>
<td>Keywords as identifiers</td>
</tr>
<tr>
<td>Data-type</td>
<td>GRAPHIC data type</td>
</tr>
<tr>
<td></td>
<td>LONGINT data type</td>
</tr>
<tr>
<td></td>
<td>NUM as a synonym for NUMERIC</td>
</tr>
<tr>
<td></td>
<td>VARGRAPHIC data type</td>
</tr>
<tr>
<td>Literal</td>
<td>G'double-byte-character-string-literal'</td>
</tr>
<tr>
<td>rowid-pseudo-column</td>
<td></td>
</tr>
<tr>
<td>Statement or component</td>
<td>CA IDMS extensions</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>SQL declaration sections</td>
<td>Support for coding delimiters across multiple lines</td>
</tr>
<tr>
<td>Special-register</td>
<td>GROUP</td>
</tr>
<tr>
<td></td>
<td>CURRENT DATE</td>
</tr>
<tr>
<td></td>
<td>CURRENT TIME</td>
</tr>
<tr>
<td></td>
<td>CURRENT TIMESTAMP</td>
</tr>
<tr>
<td></td>
<td>CURRENT DATABASE</td>
</tr>
<tr>
<td></td>
<td>CURRENT SCHEMA</td>
</tr>
<tr>
<td></td>
<td>CURRENT SQLID</td>
</tr>
<tr>
<td>Aggregate-function</td>
<td>Column-name without DISTINCT in the COUNT function</td>
</tr>
<tr>
<td>COMMIT WORK statement</td>
<td>CONTINUE parameter</td>
</tr>
<tr>
<td></td>
<td>RELEASE parameter</td>
</tr>
<tr>
<td>CREATE TABLE statement</td>
<td>IN parameter</td>
</tr>
<tr>
<td></td>
<td>COMPRESS parameter</td>
</tr>
<tr>
<td></td>
<td>ESTIMATED ROWS parameter</td>
</tr>
<tr>
<td>DECLARE CURSOR statement</td>
<td>GLOBAL parameter</td>
</tr>
<tr>
<td>FETCH statement</td>
<td>BULK parameter</td>
</tr>
<tr>
<td></td>
<td>Buffer specification in the INTO parameter</td>
</tr>
<tr>
<td>INSERT statement</td>
<td>BULK parameter</td>
</tr>
<tr>
<td>ROLLBACK WORK statement</td>
<td>RELEASE parameter</td>
</tr>
<tr>
<td>SELECT statement</td>
<td>PRESERVE parameter</td>
</tr>
<tr>
<td></td>
<td>BULK parameter</td>
</tr>
<tr>
<td>WHENEVER statement</td>
<td>Label without a colon</td>
</tr>
<tr>
<td></td>
<td>CALL parameter</td>
</tr>
</tbody>
</table>
Appendix B: Summary of Limits

Logical Data Limits

<table>
<thead>
<tr>
<th>Item</th>
<th>Maximum allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tables in a dictionary</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Views in a dictionary</td>
<td>32,767</td>
</tr>
<tr>
<td>Tables in an area</td>
<td>32,767</td>
</tr>
<tr>
<td>Indexes in an area</td>
<td>32,767</td>
</tr>
<tr>
<td>Bytes in a row</td>
<td>32,760</td>
</tr>
<tr>
<td>Columns in a table or view</td>
<td>1,024</td>
</tr>
<tr>
<td>CALC keys on a table</td>
<td>1</td>
</tr>
<tr>
<td>Indexes on a table</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Columns in a CALC, index, foreign, or sort key</td>
<td>32</td>
</tr>
<tr>
<td>Length of a CALC, index, foreign, or sort key</td>
<td>2000 bytes</td>
</tr>
</tbody>
</table>

Note: This includes length of the key columns, null indicators, and varchar length fields.

Data Type Limits

<table>
<thead>
<tr>
<th>Data type</th>
<th>Largest possible\sub1.</th>
<th>Smallest possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY</td>
<td>32,760 bytes</td>
<td>1 byte</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>32,760 bytes</td>
<td>1 byte</td>
</tr>
<tr>
<td>DATE</td>
<td>'9999-12-31'</td>
<td>'0001-01-01'</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>10\sub3.\sub1.-1</td>
<td>-(10\sub3.\sub1.-1)</td>
</tr>
<tr>
<td>Positive DOUBLE PRECISION</td>
<td>Approximately 7.2E+75</td>
<td>Approximately 5.4E-79</td>
</tr>
<tr>
<td>Negative DOUBLE PRECISION</td>
<td>Approximately -5.4E-79</td>
<td>Approximately -7.2E+75</td>
</tr>
</tbody>
</table>
### Host Variable Limits

<table>
<thead>
<tr>
<th>Data type</th>
<th>Largest possible</th>
<th>Smallest possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive FLOAT</td>
<td>Approximately 7.2E+75</td>
<td>Approximately 5.4E-79</td>
</tr>
<tr>
<td>Negative FLOAT</td>
<td>Approximately -5.4E-79</td>
<td>Approximately -7.2E+75</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>16,380 double-byte characters</td>
<td>1 double-byte character</td>
</tr>
<tr>
<td>INTEGER</td>
<td>2,147,483,647</td>
<td>-2,147,483,648</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>10^3 - 1</td>
<td>-(10^3 - 1)</td>
</tr>
<tr>
<td>Positive REAL</td>
<td>Approximately 7.2E+75</td>
<td>Approximately 5.4E-79</td>
</tr>
<tr>
<td>Negative REAL</td>
<td>Approximately -5.4E-79</td>
<td>Approximately -7.2E+75</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>32,767</td>
<td>-32,768</td>
</tr>
<tr>
<td>TIME</td>
<td>'23:59:59'</td>
<td>'00:00:00'</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>'9999-12-31.23.59.59.999999'</td>
<td>'0001-01-01.00.00.00.000000'</td>
</tr>
<tr>
<td>UNSIGNED DECIMAL</td>
<td>10^3 - 1</td>
<td>0</td>
</tr>
<tr>
<td>UNSIGNED NUMERIC</td>
<td>10^3 - 1</td>
<td>0</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>32,758 bytes</td>
<td>1 byte</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>16,379 double-byte characters</td>
<td>1 double-byte character</td>
</tr>
</tbody>
</table>

**Note:** A TIME value of 24.00.00 is accepted and treated as 00.00.00.

**Note:** The largest possible data type size is theoretical. The aggregate column size of a table cannot exceed the page size less bytes reserved for control information.

For more information, see the *CA IDMS Database Administration Guide*.

### Host Variable Limits

<table>
<thead>
<tr>
<th>Item</th>
<th>Maximum allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host and indicator variables in an SQL statement</td>
<td>2,048</td>
</tr>
<tr>
<td>Total length of the host and indicator variables described by an SQLDA</td>
<td>32,767</td>
</tr>
<tr>
<td>Item</td>
<td>Maximum allowed</td>
</tr>
<tr>
<td>---------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Length of an identifier</td>
<td>32 bytes</td>
</tr>
<tr>
<td>Length of an embedded SQL statement</td>
<td>8,192 bytes</td>
</tr>
<tr>
<td>Columns in a result table</td>
<td>1,024</td>
</tr>
<tr>
<td>Columns in a GROUP BY parameter</td>
<td>255</td>
</tr>
<tr>
<td>Total length of the columns in a GROUP BY parameter</td>
<td>32,767 bytes</td>
</tr>
<tr>
<td>UNION operands in a query expression or SELECT statement</td>
<td>31</td>
</tr>
<tr>
<td>Columns in an ORDER BY parameter</td>
<td>254</td>
</tr>
<tr>
<td>Total length of the columns in an ORDER BY parameter</td>
<td>32,767 bytes</td>
</tr>
<tr>
<td>Value expressions in a query expression</td>
<td>1,024</td>
</tr>
<tr>
<td>Table names in a query specification</td>
<td>32</td>
</tr>
<tr>
<td>Query specifications in a query expression</td>
<td>32</td>
</tr>
<tr>
<td>Dynamic parameters used in a statement</td>
<td>1024</td>
</tr>
<tr>
<td>Subqueries and user-defined function invocations in a statement</td>
<td>1024</td>
</tr>
<tr>
<td>Number of arguments in user-defined function invocations. The actual limit depends on the data types and the complexity of expressions used in the function invocation.</td>
<td>620</td>
</tr>
</tbody>
</table>
Appendix C: SQL Communication Area

SQLCA

The SQL Communication Area (SQLCA) is a data structure used to return information regarding the success or failure of an SQL request.

Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
<th>Additional information</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLCAID</td>
<td>Eye-catcher (SQLCA)</td>
<td>Initialized to SQLCA*.</td>
</tr>
<tr>
<td>SQLCODE</td>
<td>SQL error code</td>
<td>For SQLCODE values, see SQLCODE Values.</td>
</tr>
<tr>
<td>SQLCERC</td>
<td>Extended information error code</td>
<td>This field contains the reason code for error or warning conditions.</td>
</tr>
<tr>
<td>SQLCNRP</td>
<td>Number of rows processed by the SQL statement</td>
<td></td>
</tr>
<tr>
<td>SQLCNRRS</td>
<td>Number of dynamic result sets returned by a called SQL-invoked procedure</td>
<td>This is a 2-byte integer value.</td>
</tr>
<tr>
<td>SQLCSR</td>
<td>Offset into the user-provided SQL statement buffer where a syntax error was recognized</td>
<td></td>
</tr>
<tr>
<td>SQLCLNO</td>
<td>Source file line number from which the SQL statement was obtained</td>
<td>This field is maintained by the precompiler and is provided for use in forming error messages.</td>
</tr>
<tr>
<td>SQLCMCT</td>
<td>Count of messages issued for this request</td>
<td></td>
</tr>
<tr>
<td>SQLCARC</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>SQLCFJB</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>SQLCERRML</td>
<td>Length of error message text in SQLERRMC</td>
<td>This is a 256-byte field containing one or more messages. Each message is preceded by a one-byte binary field containing the length of the message text.</td>
</tr>
<tr>
<td>SQLERRMC</td>
<td>Text of the error messages</td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Meaning</td>
<td>Additional information</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>SQLSTATE</td>
<td>SQL status code</td>
<td>For SQLSTATE values, see SQLSTATE Values</td>
</tr>
</tbody>
</table>

SQLSTATE

SQLSTATE is a five-character string where CA IDMS returns the status of the last SQL statement executed. It is divided into a two-character class and a three-character subclass. Standard values are associated with each class and subclass, which minimizes the need for vendors to define their own values and makes applications more portable from one environment to another.

SQLSTATE Values

The list of SQLSTATE values that CA IDMS can return appears next. The list is divided into sections based on the class (the first 2 characters of the SQLSTATE value). Each subclass (the last 3 characters of the SQLSTATE value) is listed under its associated class.

**Standard-defined Values**

Class and subclass values beginning with the characters A-H and 0-4 are established by the SQL standard organizations.

**CA IDMS-defined Values**

Class and subclass values beginning with the characters I-Z and 5-9 are vendor-defined; in this case, they are specific to CA IDMS. Any subclass value associated with a vendor-defined class is also defined by that vendor.
SQLSTATE Values

00 Successful completion
   000 No subclass

01 Warning
   000 No subclass
   004 String data, right truncation
   00C SQL-invoked procedure returned result sets
   00D Additional result sets returned
   00E Attempt to return too many result sets
   010 Column cannot be mapped
   600 Inconsistent or invalid option
   602 Entity or association already exists
   605 Entity not defined in Catalog
   606 Invalid option for physical DDL
   607 Invalid option for DMCL
   608 Connecting to a dictionary which is missing either or
      or both of DDLCAT/DDLDML areas
   610 Database is inconsistent with request
   611 SQL routine parse error
   612 ADS compilation for an SQL routine failed
   613 Drop of SQL routine completed with warnings
   638 Warning returned from table procedure

02 No data
   000 No subclass

07 Dynamic SQL error
   000 No subclass
   001 USING clause does not match dynamic parameter specification
   002 USING clause does not match target specification
   003 Cursor specification cannot be executed
   004 USING clause required for dynamic parameters

08 Connection exception
   000 No subclass
   004 SQL-server rejected establishment of SQL-connection
   006 Connection failure

0M Invalid SQL-invoked procedure reference
   000 No subclass

0N SQL/XML Mapping Error
   000 No subclass
   001 Unmappable XML name
   002 Invalid XML character
21 Cardinality violation
  000  No subclass

22 Data Exception
  000  No subclass
  001  String data, right truncation
  002  Null value, no indicator parameter
  003  Numeric value out of range
  005  Error in assignment
  007  Invalid datetime format
  008  Datetime field overflow
  00J  Nonidentical notations with the same name
  00K  Nonidentical unparsed entities with the same name
  00L  Not an XML document
  00M  Invalid XML document
  00N  Invalid XML content
  00R  XML value overflow
  00S  Invalid comment
  00T  Invalid processing instruction
  011  Substring error
  012  Division by zero
  019  Invalid escape character

23 Constraint violation
  000  No subclass
  501  Duplicate key violation

24 Invalid cursor state
  000  No subclass

25 Invalid transaction state
  000  No subclass
  006  Read-only SQL-transaction

26 Invalid SQL statement name
  000  No subclass
28 Invalid authorization specification
   000  No subclass
   602  Entity or association already defined
   605  Entity or association not previously defined
   607  Authorization ids not specified

2C Invalid character set name
   000  No subclass

34 Invalid cursor name
   000  No subclass

37 Syntax error or access rule violation
   000  No subclass

38 External routine exception
   000  No subclass
   999  ADS dialog failed or dialog does not exist

39 External routine invocation exception
   000  No subclass

3F Invalid schema name
   000  No subclass

40 Transaction rollback
   000  No subclass
   001  Serialization failure
SQLSTATE Values

42 Syntax error or access rule violation
  000  No subclass
  500  Table not found
  501  Column not found
  502  Entity already defined
  503  Authorization failure
  504  Cursor not declared or previously declared
  505  Entity not found
  506  Invalid identifier
  507  Keyword used as identifier
  600  Invalid statement
  601  Statement not valid in this context
  603  Statement not valid for this schema
  604  Invalid data type
  606  Invalid statement option
  607  Missing statement option
  609  Invalid constraint definition
  610  Invalid number of columns

50 CA-defined errors
  000  No subclass
  002  Limit exceeded
  003  Space exceeded
  008  Internal error
  00I  Schema mismatch
  00J  Invalid entity definition
  00K  Uncategorized error
  00L  Invalid calling parameters

60 CA IDMS specific errors
  000  No subclass
  001  Problem with load module or synchronization stamps
  002  Database error
  003  Rollback failed
  004  Failure while opening or describing a received cursor
  005  Unexpected error from GET/PUT SCRATCH

64 CA IDMS Physical DDL error
  000  No subclass

6U CA IDMS Utility error
  000  No subclass
SQLCODE

SQLCODE is a field in the SQLCA, the data structure that CA IDMS uses to return information about the execution of SQL statements. After CA IDMS processes an SQL statement SQL CODE contains a value that indicates the outcome of the processing.

An application program can check the value in SQLCODE after CA IDMS processes each SQL statement and can take appropriate action based on the value.

Note: For more information about the SQLCA, see the CA IDMS SQL Programming Guide.

SQLCODE Error Values

CA IDMS returns the following values in SQLCODE:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CA IDMS successfully executed the SQL statement.</td>
</tr>
<tr>
<td>&gt;1</td>
<td>CA IDMS successfully executed the SQL statement but generated a warning in the process.</td>
</tr>
<tr>
<td>100</td>
<td>CA IDMS could find no row or no more rows to process.</td>
</tr>
<tr>
<td>-4</td>
<td>CA IDMS was unable to execute the SQL statement because errors were detected during processing. The transaction remains active.</td>
</tr>
<tr>
<td>-5</td>
<td>CA IDMS terminated the transaction abnormally to recover from a processing error. The session remains active unless it was automatically connected.</td>
</tr>
<tr>
<td>-6</td>
<td>CA IDMS has detected a condition that prevents further processing. The session is released.</td>
</tr>
<tr>
<td>-7</td>
<td>CA IDMS has detected an abnormal condition that prevents further processing. The SQL session is aborted.</td>
</tr>
</tbody>
</table>

SQLCODE and SQLCNRP Values

<table>
<thead>
<tr>
<th>Result of bulk fetch</th>
<th>SQLCODE value</th>
<th>SQLCNRP value</th>
</tr>
</thead>
<tbody>
<tr>
<td>No rows are returned</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>Result of bulk fetch</td>
<td>SQLCODE value</td>
<td>SQLCNRP value</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>At least one row is returned but fewer rows than the maximum allowed</td>
<td>100</td>
<td>Equals the number of rows returned</td>
</tr>
<tr>
<td>The number of rows returned matches the maximum allowed</td>
<td>0</td>
<td>Equals the number of rows returned</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result of bulk select</th>
<th>SQLCODE value</th>
<th>SQLCNRP value</th>
</tr>
</thead>
<tbody>
<tr>
<td>No rows are returned</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>At least one row is returned but fewer rows than the maximum allowed</td>
<td>100</td>
<td>Greater than 0 and less than or equal to the maximum allowed</td>
</tr>
<tr>
<td>The number of rows returned exceeds the maximum allowed</td>
<td>Less than 0</td>
<td>Equal to the maximum allowed</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result of bulk insert</th>
<th>SQLCODE value</th>
<th>SQLCNRP value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fewer rows than the number of rows specified are inserted because the insert failed on a row</td>
<td>Less than 0</td>
<td>Equal to the relative row number of the failing row</td>
</tr>
<tr>
<td>The number of rows inserted matches the number of rows specified</td>
<td>0</td>
<td>Equal to the number of rows inserted</td>
</tr>
</tbody>
</table>
01 SQLCA.
 02 SQLCAID PIC X(8).
 02 SQLCODE PIC S9(9) COMP.
 02 SQLSID PIC X(8).
 02 SQLINFO.
 03 S Q L C E R C PIC S9(9) COMP.
 03 FILLER PIC S9(9) COMP.
 03 SQLCERP PIC S9(9) COMP.
 03 FILLER PIC S9(9) COMP.
 03 SQLCER PIC S9(9) COMP.
 03 FILLER PIC S9(9) COMP.
 03 SQLCN0 PIC S9(9) COMP.
 03 SQLCMT PIC S9(9) COMP.
 03 SQLCARG PIC S9(9) COMP.
 03 SQLCMB PIC S9(9) COMP.
 03 FILLER PIC S9(9) COMP.
 03 FILLER PIC S9(9) COMP.
 02 SQLCINF2 REDEFINES SQLINFO.
 03 SQLERRD PIC S9(9) COMP OCCURS 12.
 02 SQLMSG.
 03 SQLCERL PIC S9(9) COMP.
 03 SQLERR PIC X(256).
 02 SQLMSG2 REDEFINES SQLMSG.
 03 FILLER PIC X(2).
 03 SQLERRM.
 04 SQLCERML PIC S9(4) COMP.
 04 SQLERM COMP X(256).
 02 SQLSTATE PIC X(5).
 02 SQLCNF PIC X(1).
 02 SQLNRRS PIC S9(4) COMP.
 02 FILLER PIC X(8).
 02 SQLWORK PIC X(16).
 02 SQLW R K 2 REDEFINES SQLWORK.
 03 SQLERRP.
 04 SQLCVAL PIC X(5). Included by the
 04 FILLER PIC X(3).
 04 SQLWARN. precompiler for
 04 SQLWARN0 PIC X(1).
 04 SQLWARN1 PIC X(1).
 04 SQLWARN2 PIC X(1).
 04 SQLWARN3 PIC X(1).
 04 SQLWARN4 PIC X(1).
 04 SQLWARN5 PIC X(1).
 04 SQLWARN6 PIC X(1).
 04 SQLWARN7 PIC X(1).
DECLARE 1  SQLCA,
  2   SQLCAID  CHARACTER (8),
  2   SQLCODE  FIXED BINARY (31),
  2   SQLCSID  CHARACTER (8),
  2   SQLINFO,
  3   SQLCERC  FIXED BINARY (31),
  3   FILLER
    2   SQLCINFO,
    2   SQLCERC  FIXED BINARY (31),
    2   FILLER
    3   SQLCINFO,
    3   SQLCERC  FIXED BINARY (31),
    3   FILLER
  2   SQLCINF2 BASED (ADDR(SQLCINFO)),
  2   SQLERRD  FIXED BINARY (31),
  2   SQLMSG,
  3   SQLCERL  FIXED BINARY (31),
  3   SQLCERM  CHARACTER (256),
  2   SQLSTATE  CHARACTER (5),
  2   SQLCNF  CHARACTER (1),
  2   SQLCNRS  FIXED BINARY (15),
  2   SQLCNR  CHARACTER (8),
  2   SQLCVAL  CHARACTER (5),
  3   FILLER
  2   SQLCWRK2 BASED(ADDR(SQLWORK)),
  2   SQLERRP,
  3   SQLCVAL  CHARACTER (5),
  3   FILLER
    2   SQLCWRK2 BASED(ADDR(SQLWORK)),
    2   SQLERRP,
    3   SQLCVAL  CHARACTER (5),
    3   FILLER
  2   SQLCINF2 BASED (ADDR(SQLCINFO)),
  2   SQLERRD  FIXED BINARY (31),
  2   SQLMSG,
  3   SQLCERL  FIXED BINARY (31),
  3   SQLCERM  CHARACTER (256),
  2   SQLSTATE  CHARACTER (5),
  2   SQLCNF  CHARACTER (1),
  2   SQLCNRS  FIXED BINARY (15),
  2   SQLCNR  CHARACTER (8),
  2 SQLWORK CHARACTER (16);
## Appendix D: SQL Descriptor Area

### SQLDA

The SQL Descriptor Area (SQLDA) is a data structure used to describe variable data passed as part of a dynamic SQL statement.

### SQLDA Fields

The SQLDA consists of the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLDAID</td>
<td>CHARACTER(8)</td>
<td>Set to SQLDA* on a DESCRIBE</td>
</tr>
<tr>
<td>SQLN</td>
<td>INTEGER</td>
<td>Maximum number of SQLVAR occurrences</td>
</tr>
</tbody>
</table>
| SQLD    | INTEGER     | Actual number of SQLVAR occurrences:
|         |             | - 0—Not a SELECT statement                                              |
|         |             | - 1 through SQLN—Number of columns                                       |
|         |             | - Greater than SQLN—Not enough SQLVAR entries                            |
| SQLVAR  |             | Structure occurring SQLN times                                          |

### SQLVAR Fields

The structure SQLVAR in the SQLDA consists of the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| SQLLEN  | INTEGER     | Length
|         |             | (Additional information provided under "SQLLEN")                       |
| SQLTYPE | SMALLINT    | Column data type
|         |             | (Additional information provided under "SQLTYPE")                      |
### Field, Data type, Meaning

<table>
<thead>
<tr>
<th>Field</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLSCALE</td>
<td>SMALLINT</td>
<td>Scale (for exact numeric data types) (Additional information provided under &quot;SQLSCALE&quot;)</td>
</tr>
<tr>
<td>SQLPRECISION</td>
<td>SMALLINT</td>
<td>Precision (Additional information provided under &quot;SQLPRECISION&quot;)</td>
</tr>
<tr>
<td>SQLALN</td>
<td>SMALLINT</td>
<td>Data alignment flag (Additional information provided under &quot;SQLALN and SQLNALN &quot;)</td>
</tr>
<tr>
<td>SQLNALN</td>
<td>SMALLINT</td>
<td>Null indicator alignment flag (Additional information provided under &quot;SQLALN and SQLNALN &quot;)</td>
</tr>
<tr>
<td>SQLNULL</td>
<td>SMALLINT</td>
<td>Length of null indicator (Additional information provided under &quot;SQLNULL&quot;)</td>
</tr>
<tr>
<td>SQLNAME</td>
<td>CHARACTER(32)</td>
<td>Column name</td>
</tr>
</tbody>
</table>

### Notes

The SQLDA can be used by an application program in the following ways:

- As output on a DESCRIBE or PREPARE statement as the location into which the DBMS returns the descriptions of selected columns.
- As input on a FETCH statement to describe the target area for selected columns.

### SQLLEN

The definition of SQLLEN by data type is:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Definition of SQLLEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>Number of characters</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>Maximum number of characters</td>
</tr>
<tr>
<td>DATETIME</td>
<td>Length of the date/time column</td>
</tr>
<tr>
<td>BINARY</td>
<td>Number of bytes</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>Number of two-byte characters</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>Maximum number of two-byte characters</td>
</tr>
</tbody>
</table>
SQLTYPE

A code indicating the data type of the host variable. The codes from 1 through 127 are the same as those defined in the SQL standard. Codes greater than 127 were assigned before SQL standards existed for these datatypes.

Current SQL standard codes are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>2</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>3</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>4</td>
<td>INTEGER (four bytes)</td>
</tr>
<tr>
<td>5</td>
<td>SMALLINT (two bytes)</td>
</tr>
<tr>
<td>6</td>
<td>FLOAT</td>
</tr>
<tr>
<td>7</td>
<td>REAL</td>
</tr>
<tr>
<td>8</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>9</td>
<td>Datetime (DATE, TIME, or TIMESTAMP)</td>
</tr>
<tr>
<td>10</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>Reserved</td>
</tr>
<tr>
<td>12</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>

Current CA IDMS extensions are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>NUMERIC (UNSIGNED)</td>
</tr>
<tr>
<td>129</td>
<td>DECIMAL (UNSIGNED)</td>
</tr>
<tr>
<td>130</td>
<td>BIGINT or LONGINT (eight bytes)*</td>
</tr>
<tr>
<td>131</td>
<td>Reserved</td>
</tr>
<tr>
<td>132</td>
<td>GRAPHIC</td>
</tr>
<tr>
<td>133</td>
<td>VARGRAPHIC</td>
</tr>
<tr>
<td>134</td>
<td>Reserved for precompiler use (GROUP)</td>
</tr>
<tr>
<td>135</td>
<td>BINARY/SQLBIN*</td>
</tr>
<tr>
<td>136</td>
<td>TID or Tuple ID (eight bytes, used for ROWID)</td>
</tr>
</tbody>
</table>
### SQLSCALE

The definition of SQLSCALE by data type is:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Definition of SQLSCALE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact numeric</td>
<td>Scale.</td>
</tr>
<tr>
<td>Datetime</td>
<td>Code indicating the precision of the date/time value. Permissible values in the first implementation are:</td>
</tr>
<tr>
<td></td>
<td>■ 1 for DATE</td>
</tr>
<tr>
<td></td>
<td>■ 2 for TIME</td>
</tr>
<tr>
<td></td>
<td>■ 3 for TIMESTAMP</td>
</tr>
</tbody>
</table>

### SQLPRECISION

The definition of SQLPRECISION by data type is:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Definition of SQLPRECISION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact numeric</td>
<td>Precision.</td>
</tr>
<tr>
<td>Approximate numeric</td>
<td>Precision.</td>
</tr>
<tr>
<td>Datetime</td>
<td>Precision of the fraction field (0 or 6), if applicable. If passed on a dynamic FETCH statement, SQLPRECISION is ignored for date/time data types.</td>
</tr>
</tbody>
</table>
SQLALN and SQLNALN

SQLALN and SQLNALN are flags that indicate whether data (SQLALN) and the null indicator (SQLNALN) are aligned:

- 0—Not aligned.
- 1—Aligned based on natural alignment rules for the platform. For example, on both the System/370 and VMS environments, the natural alignment for the INTEGER data type is on a four-byte boundary.

On a DESCRIBE statement, the SQLALN and SQLNALN fields are set to 0.

SQLNULL

CA IDMS defines SQLNULL to be:

- 0—No null indicator
- 1, 2, 4—Null indicator length

A DESCRIBE statement returns 4 if nulls are allowed and 0 otherwise.
Appendix E: SYSTEM Tables and SYSCA Views

Overview

The catalog component of the dictionary comprises SYSTEM tables. SYSTEM tables contain logical definition information for a database defined with SQL DDL and information on the definition of the physical implementation of the database.

SYSCA views are views defined on a subset of SYSTEM tables. SYSCA views restrict information that the user can select from SYSTEM tables to data about tables for which the user holds SELECT privilege.

Important: SYSTEM tables are defined in segment SYSCAT. This segment is created as part of defining the SYSTEM schema. SYSCAT should not be used as a segment name, and no attempt should be made to alter or delete the SYSCAT segment definition.

SYSTEM.AM

Description

A row of SYSTEM.AM identifies an access module.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Access module name</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>VERSION</td>
<td>Access module version</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SCHEMA</td>
<td>Access module schema</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CTIME</td>
<td>Date and time the access module was created</td>
<td>TIMESTAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>LENGTH</td>
<td>Length, in bytes, of the access module</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FILLER</td>
<td>Reserved for future use</td>
<td>BINARY(20)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
### SYSTEM.AMDEP

**Description**

A row of SYSTEM.AMDEP identifies a table or view referenced in an access module.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Access module name.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>VERSION</td>
<td>Access module version.</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TABSCHEMA</td>
<td>Schema-name qualifier of the table.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TABLE</td>
<td>Table or view name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>STAMP</td>
<td>Date and time the table was created or last altered.</td>
<td>TIMESTAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TYPE</td>
<td>Type of table:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ T—Base table</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ V—View</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FILLER</td>
<td>Reserved for future use.</td>
<td>BINARY(25)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>

### SYSTEM.AREA

**Description**

A row of SYSTEM.AREA represents an area within a segment.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEGMENT</td>
<td>Segment name.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NAME</td>
<td>Area name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CTIME</td>
<td>Date and time when the area was created.</td>
<td>TIMESTAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UTIME</td>
<td>Date and time when the area was last updated.</td>
<td>TIMESTAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CRITTIME</td>
<td>Date and time of the last critical change to the area.</td>
<td>TIMESTAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Definition date/time stamp for table validation.</td>
<td>TIMESTAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CUSER</td>
<td>User ID of user who created the area.</td>
<td>TIMESTAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UUSER</td>
<td>User ID of user who last updated the area.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>-------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>TYPE</td>
<td>Area type:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ N—Non-SQL area</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ R—SQL area</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STAMPLEVEL</td>
<td>Stamp indicator:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ N—No stamp checking for a non-SQL area</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ T—Table-level stamping for an SQL area</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ S—Area-level stamping for an SQL area</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NUMFILEMAPS</td>
<td>Number of filemaps in the area.</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMSYMBOLICS</td>
<td>Number of symbolics in the area.</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DISPLACEMENT</td>
<td>Cluster displacement.</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>PAGEGROUP</td>
<td>The page group associated with the area.</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>LOWPAGE</td>
<td>Low page number of the area.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>HIGHPAGE</td>
<td>High page number of the area.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CALCHIGHPAGE</td>
<td>Primary (calc) high page number of the area.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>MAXHIGHPAGE</td>
<td>Maximum high page number of the area.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMPAGES</td>
<td>Number of pages</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>PAGESIZE</td>
<td>Size, in bytes, of each page in the area.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>PAGERESERVE</td>
<td>Page reserve size, in bytes. This column indicates the number of bytes left unused on a page when new rows are stored so that space is available for the expansion of variable-length rows during update operations.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>ORIGPAGESIZE</td>
<td>Original page size of the area.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMPAGESUSED</td>
<td>Number of nonempty pages in the area.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMROWS</td>
<td>Number of rows in the area.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>PCTSPACEUSED</td>
<td>Percent of space used in the area.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FILLER</td>
<td>Reserved for future use.</td>
<td>BINARY(40)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
A row of SYSTEM.BUFFER represents a DMCL buffer.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMCL</td>
<td>DMCL name.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NAME</td>
<td>Buffer name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TYPE</td>
<td>Buffer type:</td>
<td>CHAR(2)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>▪ BC—Definition represents a standard buffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>▪ JB—Definition represents a journal buffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTIME</td>
<td>Date and time when the DMCL buffer was created.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UTIME</td>
<td>Date and time when the DMCL buffer was last updated.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CRITTIME</td>
<td>Date and time of the last critical change made to the DMCL buffer.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CUSER</td>
<td>User ID of user who created the DMCL buffer.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UUSER</td>
<td>ID of the user who last updated the DMCL buffer.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>PAGESIZE</td>
<td>Size, in bytes, of pages in the buffer. For native VSAM data sets, this column indicates the control interval size.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>LOCALPAGES</td>
<td>Number of pages in the local mode buffer.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CVPAGES</td>
<td>Initial number of pages in the central version mode buffer.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>MAXPAGES</td>
<td>Maximum number of pages in the central version mode buffer.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
## SYSTEM.COLUMN

### Description

A row of SYSTEM.COLUMN represents a column in a table.

View SYSCA.COLUMN is defined on SYSTEM.COLUMN.

**Note:** For more information, see SYSCA Objects.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Column name.</td>
<td>CHAR(32)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMBER</td>
<td>Relative number of the column within the table.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>

---

### Appendix E: SYSTEM Tables and SYSCA Views

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEYLENGTH</td>
<td>The maximum sort-key or CALC-key length for native VSAM files.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>BUFNI</td>
<td>Number of I/O buffers in the native VSAM non-shared resources (NSR) buffer that are used for index entries.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>STRNO</td>
<td>Maximum number of concurrent requests permitted against an area assigned to the native VSAM buffer.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>STGFLAG</td>
<td>Storage location indicator.</td>
<td>BINAR Y(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FLAG</td>
<td>Buffer flag; X'80'—Native VSAM LSR or NSR buffer</td>
<td>BINAR Y(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FILLER</td>
<td>Reserved for future use.</td>
<td>BINAR Y(40)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------</td>
<td>-----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>SCHEMA</td>
<td>Schema-name qualifier of the table or view that contains the column.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TABLE</td>
<td>Name of the table or view that contains the column.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TYPE</td>
<td>Column data type in character format.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TYPECODE</td>
<td>Column data type in numeric format. For a description of values in this field, see SQLTYPE.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>PRECISION</td>
<td>Precision of numeric columns.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SCALE</td>
<td>Scale of exact numeric fields or code indicating the type of date/time column. For a description of values in this field, see SQLSCALE.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
| NULLS      | Nulls allowed:  
  - N—No  
  - Y—Yes | CHAR(1) | NOT NULL |
| DEFAULT    | Default value stored:  
  - N—No  
  - Y—Yes | | |
| VOFFSET    | Offset to column value within a row. | SMALL INT | NOT NULL |
| VLENGTH    | Length of column value. | SMALL INT | NOT NULL |
| NOFFSET    | Offset to NULL indicator for the column a row. | SMALL INT | NOT NULL |
| NLENGTH    | Length of NULL indicator. | SMALL INT | NOT NULL |
| NUMVALUES  | If the column is the first column in an index key, the number of unique values in the column when statistics were last updated. | INTEGER | NOT NULL |
### SYSTEM.CONSTKEY

#### Description

A row of SYSTEM.CONSTKEY represents the foreign key in a referential constraint defined with a CREATE CONSTRAINT statement.

View SYSCA.CONSTKEY is defined on SYSTEM.CONSTKEY.

**Note:** For more information, see SYSCA Objects.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMA</td>
<td>Schema-name qualifier of the constraint.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NAME</td>
<td>Constraint name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SEQUENCE</td>
<td>Key column sequence number.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
### SYSTEM.CONSTRAINT

**Description**

A row of SYSTEM.CONSTRAINT represents a referential constraint defined with a CREATE CONSTRAINT statement.

View SYSCA.CONSTRAINT is defined on SYSTEM.CONSTRAINT.

**Note:** For more information, see SYSCA Objects.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMA</td>
<td>Schema-name qualifier of the constraint.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NAME</td>
<td>Constraint name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>REFSHEMA</td>
<td>Schema-name qualifier of the referenced table.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>REFTABLE</td>
<td>Name of the referenced table.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TABSCHEMA</td>
<td>Schema-name qualifier of the referencing table.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TABLE</td>
<td>Name of the referencing table.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------------------------------------------------------------------</td>
<td>------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>CTIME</td>
<td>Date and time the constraint was created.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMCOLUMNS</td>
<td>Number of columns in the constraint (foreign key).</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>REFCOLUMNS</td>
<td>Referenced table key column number array. Each eight bits contains the relative column number of a referenced column (for NUMCOLUMNS entries).</td>
<td>BINAR Y(64)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>COLUMNS</td>
<td>Referencing table foreign key column number array. Each eight bits contains the relative column number of a referencing column (for NUMCOLUMNS entries).</td>
<td>BINAR Y(64)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMSORTCOLS</td>
<td>Number of columns in the sort key.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SORTCOLUMNS</td>
<td>Sort key column number array. Each eight bits contains the relative column number of a sort column (for NUMSORTCOLS entries).</td>
<td>BINAR Y(64)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SORTORDER</td>
<td>Sort order indicator. Each byte indicates the order of a sort column (for NUMSORTCOLS entries):</td>
<td>CHAR(32)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ A—Ascending</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ D—Descending</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLUSTER</td>
<td>Referencing table cluster indicator:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ Y—Clustered</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ N—Not clustered</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNIQUE</td>
<td>Uniqueness indicator for sort key:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ Y—Unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ N—Not unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------</td>
<td>-----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>TYPE</td>
<td>Type of constraint:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ L—Linked</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ U—Unlinked</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ X—Linked indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPRESS</td>
<td>Index keys compressed:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ Y—Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ N—No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IXBLKLENGTH</td>
<td>Index block (SR8) length.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>IXBLKCONTAINS</td>
<td>Number of keys in index block.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DISPLACEMENT</td>
<td>Index displacement (the number of pages by which bottom-level SR8 records are displaced from the referenced row).</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>REFNEXT</td>
<td>Offset to next db-key pointer within the referenced table.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>REFPRIOR</td>
<td>Offset to prior db-key pointer within the referenced table.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NEXT</td>
<td>Offset to next db-key pointer within the referencing table.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>PRIOR</td>
<td>Offset to prior db-key pointer within the referencing table.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>OWNER</td>
<td>Offset to owner db-key pointer within the referencing table.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMSETS</td>
<td>Number of referenced rows when statistics were last updated.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGMEMROWS</td>
<td>Average number of referencing rows per referenced row when statistics were last updated.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>LONGESTMEM</td>
<td>Highest number of referencing rows per referenced row when statistics were last updated.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>-----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>SECLONGMEM</td>
<td>Second highest number of referencing rows per referenced row when statistics were last updated.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMLONGMEM</td>
<td>Number of referenced rows having LONGESTMEM referencing rows when statistics were last updated.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGMEMPAGE</td>
<td>Average number of database pages containing referencing rows per referenced row when statistics were last updated. Rows accounted for in AVGMEMPAGE (above) are not included in this average.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>MAXMEMPAGE</td>
<td>The number referenced rows whose referencing occupied more than 20 database pages when statistics were last updated. Rows accounted for in AVGMEMPAGE (above) are not included in this number.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGMEMCLUSCNT</td>
<td>Average number of I/Os required to read all referencing rows associated with a referenced row when statistics were last updated, if one buffer page was available. This count includes I/O to read the bottom-level SR8.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGBMEMCLUSCNT</td>
<td>Average number of I/Os required to read all referencing rows associated with a referenced row when statistics were last updated, if three buffer pages were available. This count includes I/O to read the bottom-level SR8.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>AVGCMEMCLUSCNT</td>
<td>Average number of I/Os required to read all referencing rows associated with a referenced row when statistics were last updated, if five buffer pages were available. This count includes I/O to read the bottom-level SR8.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGDGMEMCLUSCNT</td>
<td>Average number of I/Os required to read all referencing rows associated with a referenced row when statistics were last updated, if 10 buffer pages were available. This count includes I/O to read the bottom-level SR8.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGEMEMCLUSCNT</td>
<td>Average number of I/Os required to read all referencing rows associated with a referenced row when statistics were last updated, if 20 buffer pages were available. This count includes I/O to read the bottom-level SR8.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGSR8ROWS</td>
<td>Average number of SR8s per referenced row in a linked indexed constraint when statistics were last updated.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>LONGESTSR8</td>
<td>Highest number of bottom-level SR8s per referenced row in a linked indexed constraint when statistics were last updated.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SECLONGSR8</td>
<td>Second highest number of bottom-level SR8s per referenced row in a linked indexed constraint when statistics were last updated.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMLONGSR8</td>
<td>Number of referenced rows in a linked indexed constraint having LONGESTSR8 SR8s when statistics were last updated.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------</td>
<td>-------------------</td>
</tr>
<tr>
<td>AVGSR8PAGES</td>
<td>Average number of pages containing bottom-level SR8s per referenced row in a linked indexed constraint when statistics were last updated. This average does not include pages accounted for in MAXSR8PAGES.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>MAXSR8PAGES</td>
<td>The number of referenced rows in a linked indexed constraint when statistics were last updated for which the number of pages containing bottom-level SR8s is greater than 20.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGSR8LEAFS</td>
<td>Average number of bottom-level SR8s per referenced row in a linked indexed constraint when statistics were last updated.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGSR8LEVELS</td>
<td>The average high level number per referenced row in a linked indexed constraint when statistics were last updated. Level number refers to the number of levels above the bottom level.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGASR8CLUSCNT</td>
<td>Average number of I/Os required to read all SR8s associated with a referenced row in a linked indexed constraint when statistics were last updated, if one buffer page was available.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AGBSR8CLUSCNT</td>
<td>Average number of I/Os required to read all SR8s associated with a referenced row in a linked indexed constraint when statistics were last updated, if three buffer pages were available.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
### SYSTEM.DBNAME

#### Description

DBNAME contains information about an entry in a database name table.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBTABLE</td>
<td>Name of database name table.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NAME</td>
<td>Database name entry in the database name table. If this column contains *DEFAULT, the row represents the default subschema mapping for the database name table.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DMCL</td>
<td>Reserved for future use.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
### System.DBName

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE</td>
<td>Reserved for future use.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>LOADLIST</td>
<td>Reserved for future use.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FLAG</td>
<td>Database name flag:</td>
<td>BINAR Y(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>- X'80'—Match on subschema name is required</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- X'40'—Mixed Page Groups are allowed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- X'20'—Mixed Page Groups Verify is ON</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- X'01'—DB Group Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTIME</td>
<td>Date and time when the database name was created.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UTIME</td>
<td>Date and time when the database name was last updated.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CUSER</td>
<td>ID of the user who created the database name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UUSER</td>
<td>ID of the user who last updated the database name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>GROUPFLAG</td>
<td>DB Group flag:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>- E—Enabled at startup</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- D—Disabled at startup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NUMSEGMENTS</td>
<td>Number of segments associated with the database name.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMSUBSCHEMAS</td>
<td>Number of subschemas associated with the database name.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>USAGEFLAG</td>
<td>Indicates general use or utility use only. X'80' DBName for utility use only.</td>
<td>BINAR Y(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FILLER</td>
<td>Reserved for future use.</td>
<td>BINAR Y(17)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
## SYSTEM.DBSEGMENT

### Description

DBSEGMENT associates the name of a segment with a database name in a database name table.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBTABLE</td>
<td>Database table name.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DBNAME</td>
<td>Database name entry in the database table.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NAME</td>
<td>Segment name.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FILLER</td>
<td>Reserved for future use.</td>
<td>BINARY(20)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>

## SYSTEM.DBSSC

### Description

DBSSC associates a subschema mapping with a database name in a database name table.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBTABLE</td>
<td>Name of the database table.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DBNAME</td>
<td>Database name entry in the database table.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FROMSSC</td>
<td>Subschema name passed at run unit signon (FROM subschema).</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TOSSC</td>
<td>Name of the subschema to which the passed subschema name maps (TO subschema).</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FOREIGNDBNAME</td>
<td>Database name to be accessed if no database name is specified.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
## SYSTEM.DBTABLE

### Description

DBTABLE contains information about the database name table.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Name of the database name table.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CVSYSTEM</td>
<td>Reserved for future use.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CTIME</td>
<td>Date and time when the database name table was created.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UTIME</td>
<td>Date and time when the database name table was last updated.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CUSER</td>
<td>ID of the user who created the database name table.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UUSER</td>
<td>ID of the user who last updated the database name table.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMDBNAMES</td>
<td>Number of names in the database name table.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMSEGMENTS</td>
<td>Number of segments in the database name table.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMSUBSCHEMAS</td>
<td>Number of subschemas in the database name table.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMGROUPS</td>
<td>Number of DB Groups in the database name table.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>

### Note:

1. A TOSSC value of spaces and a FOREIGNDBNAME of *DEFAULT indicates all matching subschemas should use DBNAME mapping rules.
## System.Dmcl

**Description**

A row of System.Dmcl contains information about a DMCL.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Description</th>
<th>Data Type</th>
<th>Null Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>DMCL name.</td>
<td>Char(8)</td>
<td>Not Null</td>
</tr>
<tr>
<td>CTime</td>
<td>Date and time when the DMCL was created.</td>
<td>Times Tamp</td>
<td>Not Null</td>
</tr>
<tr>
<td>UTime</td>
<td>Date and time when the DMCL was last updated.</td>
<td>Times Tamp</td>
<td>Not Null</td>
</tr>
<tr>
<td>CritTime</td>
<td>Date and time of the last critical change made to the DMCL.</td>
<td>Times Tamp</td>
<td>Not Null</td>
</tr>
<tr>
<td>CUser</td>
<td>ID of the user who created the DMCL.</td>
<td>Char(18)</td>
<td>Not Null</td>
</tr>
<tr>
<td>UUser</td>
<td>ID of the user who last updated the DMCL.</td>
<td>Char(18)</td>
<td>Not Null</td>
</tr>
<tr>
<td>Buffer</td>
<td>Default buffer for the DMCL.</td>
<td>Char(18)</td>
<td>Not Null</td>
</tr>
<tr>
<td>DbTable</td>
<td>Database name table for the DMCL.</td>
<td>Char(8)</td>
<td>Not Null</td>
</tr>
<tr>
<td>NumBuffers</td>
<td>Number of buffers defined in the DMCL.</td>
<td>Small Int</td>
<td>Not Null</td>
</tr>
<tr>
<td>NumJrnLBuffers</td>
<td>Number of journal buffers defined in the DMCL.</td>
<td>Small Int</td>
<td>Not Null</td>
</tr>
<tr>
<td>NumJournals</td>
<td>Number of journals defined in the DMCL.</td>
<td>Small Int</td>
<td>Not Null</td>
</tr>
<tr>
<td>SharedCache</td>
<td>Default shared cache.</td>
<td>Char(16)</td>
<td>Not Null</td>
</tr>
</tbody>
</table>
### SYSTEM.DMCLAREA

**Description**

A row of SYSTEM.DMCLAREA contains information about an area whose segment has been included in the DMCL.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMCL</td>
<td>DMCL name.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SEGMENT</td>
<td>Name of the segment that contains the area.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NAME</td>
<td>Area name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CTIME</td>
<td>Date and time when the area was created.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UTIME</td>
<td>Date and time when the area was last updated.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------</td>
<td>-----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>CRITTIME</td>
<td>Date and time of the last critical change made to the area.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CUSER</td>
<td>ID of the user who created the area.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UUSER</td>
<td>ID of the user who last updated the area.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>STARTUP</td>
<td>Startup indicator. This indicates the READY action to be taken when the system is started following an orderly shutdown. Values are:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ U—Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ R—Retrieval</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ T—Transient retrieval</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ X—Set status offline</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WARMSTART</td>
<td>Warmstart indicator. This indicates the READY action to be taken when the system is started following an abnormal termination. Values are:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ U—Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ R—Retrieval</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ T—Transient retrieval</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ X—Set status offline</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ C—Maintain current status</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAGERESERVE</td>
<td>Page reserve. Number of bytes to be left unused on a page when new rows are stored on pages in the area. For this DMCL, the page reserve specification overrides the page reserve specification in the segment's area definition.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
# SYSTEM.DMCLFILE

## Description

A row of SYSTEM.DMCLFILE contains information about a file override that has been included in the DMCL.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMCL</td>
<td>DMCL name.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SEGMENT</td>
<td>Name of the segment associated with the file.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NAME</td>
<td>File name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CTIME</td>
<td>Date and time when the file was created.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UTIME</td>
<td>Date and time when the file was last updated.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CRITTIME</td>
<td>Date and time of the last critical change made to the file.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CUSER</td>
<td>ID of the user who created the file.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UUSER</td>
<td>ID of the user who last updated the file.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------</td>
<td>-----------</td>
<td>-------------------</td>
</tr>
<tr>
<td>BUFFER</td>
<td>Name of the buffer defined within the DMCL which is used by the file. This specification overrides the default buffer established for the segment included in the DMCL and the default buffer established for the DMCL.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DDNAME</td>
<td>Depending on the operating system, DDName (z/OS), filename (z/VSE), or linkname of the file. This specification overrides the specification in the segment's file definition.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DISP</td>
<td>Dataset disposition (IBM) or shared update). This specification overrides the segment's file definition.</td>
<td>CHAR(4)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
| DATASPACE   | z/OS memory cache indicator:  
  ■ N—Files will not use memory cache (Z-Storage or dataspace)  
  ■ Y—Files will use memory cache (Z-Storage or dataspace) | CHAR(1) | NOT NULL |
| ESAREAD     | Reserved for future use. | CHAR(1) | NOT NULL |
| ESAPRELOAD  | Reserved for future use. | CHAR(1) | NOT NULL |
| SHAREDCACHE | Shared cache:  
  ■ NO—Not used  
  ■ AVAILABLE—Choose first available  
  ■ Cache Name—Name of the cache to be used | CHAR(16) | NOT NULL |
| FILLER      | Reserved for future use. | BINAR Y(21) | NOT NULL |
**SYSTEM.DMCLSEGMENT**

**Description**

A row of SYSTEM.DMCLSEGMENT contains information about a segment definition and a DMCL where the segment has been included. DMCLSEGMENT contains information specific to the DMCL where the segment has been included.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMCL</td>
<td>DMCL name.</td>
<td>CHAR( 8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NAME</td>
<td>Segment name.</td>
<td>CHAR( 8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CTIME</td>
<td>Date and time when the segment was created.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UTIME</td>
<td>Date and time when the segment was last updated.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CRITTIME</td>
<td>Date and time of the last critical change made to the segment.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CUSER</td>
<td>ID of the user who created the segment.</td>
<td>CHAR( 18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UUSER</td>
<td>ID of the user who last updated the segment.</td>
<td>CHAR( 18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>BUFFER</td>
<td>Name of the buffer within the DMCL which is the default for all files defined within the segment unless specifically overridden.</td>
<td>CHAR( 18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>STARTUP</td>
<td>Startup indicator. This indicates the READY action to be taken when the system is started following an orderly shutdown. Values are: U—Update, R—Retrieval, T—Transient retrieval, X—Set status offline.</td>
<td>CHAR( 1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>---------------</td>
<td>--------------------</td>
</tr>
</tbody>
</table>
| WARMSTART  | Warmstart indicator. This indicates the READY action to be taken when the system is started following an abnormal termination. Values are:  
  ■ U—Update  
  ■ R—Retrieval  
  ■ T—Transient retrieval  
  ■ X—Set status offline  
  ■ C—Maintain current status | CHAR(1)       | NOT NULL           |
| DATASHARE  | Data sharing indicator.  
  ■ 'Y' - all areas in the segment are eligible to be shared for update  
  ■ 'N' - no areas in the segment are shared | CHAR(1)       | NOT NULL           |
| SHAREDCACHE| Default shared cache.                                                               | CHAR(16)      | NOT NULL           |
| FILLER     | Reserved for future use.                                                            | CHAR(23)      | NOT NULL           |

**SYSTEM.FILE**

**Description**

A row of SYSTEM.FILE represents a file associated with a segment.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEGMENT</td>
<td>Segment name.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NAME</td>
<td>File name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CTIME</td>
<td>Date and time when the file was created.</td>
<td>TIMES TAMPP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------</td>
<td>-----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>UTIME</td>
<td>Date and time when the file was last updated.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CRITTIME</td>
<td>Date and time of the last critical change made to the file.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CUSER</td>
<td>The ID of the user who created the file.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UUSER</td>
<td>The ID of the user who last updated the file.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMFILEMAPS</td>
<td>The number of area page ranges mapped to the file.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>BLOCKSIZE</td>
<td>Block size, in bytes, of the file. This is the largest page size</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DDNAME</td>
<td>Depending on the operating system, DDName (z/OS), or filename</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>ACCESSMETHOD</td>
<td>File access method.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>VMUSERID</td>
<td>In the z/VM environment, the user ID associated with the file.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>VMVIRTADDR</td>
<td>In the z/VM environment, the virtual address of the file.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FLAG</td>
<td>File flag:</td>
<td>BINAR Y(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>□ X'80'—Native VSAM FOR CALC file</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>□ X'40'—Native VSAM FOR SET file</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVSAMSET</td>
<td>Native VSAM KSDS or PATH set name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DSNNAME</td>
<td>Dataset name.</td>
<td>CHAR(54)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DISP</td>
<td>Dataset disposition (IBM).</td>
<td>CHAR(4)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FILLER</td>
<td>Reserved for future use.</td>
<td>BINAR Y(39)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
SYSTEM.FILEMAP

Description

A row of SYSTEM.FILEMAP relates page ranges of an area of a segment to block ranges of a file in the same segment (area-to-file mapping).

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEGMENT</td>
<td>Segment name.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AREA</td>
<td>Area name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FILE</td>
<td>File name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>PAGESIZE</td>
<td>Size, in bytes, of each page in the area.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>LOWPAGE</td>
<td>Low page number of the area page range.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>HIGHPAGE</td>
<td>High page number of the area page range.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>LOWBLOCK</td>
<td>Low relative block number (RBN) of the range of file blocks to which the area page range is mapped.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>HIGHBLOCK</td>
<td>High relative block number (RBN) of the range of file blocks to which the area page range is mapped.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>ACCESSMETHOD</td>
<td>File access method.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FLAG</td>
<td>File flag:</td>
<td>BINAR Y(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>▪ X’80’—Native VSAM FOR CALC file</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>▪ X’40’—Native VSAM FOR SET file</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FILLER</td>
<td>Reserved for future use.</td>
<td>BINAR Y(39)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
**SYSTEM.INDEX**

**Description**

A row of SYSTEM.INDEX represents an index that has been defined on a table with a CREATE INDEX statement.

View SYSCA.INDEX is defined on SYSTEM.INDEX.

**Note:** For more information, see SYSCA Objects.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Index name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SCHEMA</td>
<td>Schema-name qualifier for the indexed table.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TABLE</td>
<td>Name of the indexed table.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SEGMENT</td>
<td>Segment containing the area where index entries are stored.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AREA</td>
<td>Area where index entries are stored.</td>
<td>CHAR(18)</td>
<td></td>
</tr>
<tr>
<td>INDEXID</td>
<td>Internal index ID number. Index IDs are automatically assigned to each index and are unique within the index area.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CTIME</td>
<td>Date and time when the index was created.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMCOLUMNS</td>
<td>Number of columns in the index key.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>IXCOLUMNS</td>
<td>Internal index key column number array. This array consists of 32 SMALLINT bytes.</td>
<td>BINARY(64)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>IXORDERS</td>
<td>Sort order indicator array consisting of 32 CHAR(1) bytes. Values are:</td>
<td>CHAR(32)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ A—Ascending</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ D—Descending</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------------</td>
<td>-----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>UNIQUE</td>
<td>Unique key indicator:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>▪ Y—Unique index</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>▪ N—Not unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLUSTER</td>
<td>Cluster indicator:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>▪ Y—Clustered index</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>▪ N—Not clustered</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPRESS</td>
<td>Index entry compression indicator:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>▪ Y—Compressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>▪ N—Not compressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IXBLKLENGTH</td>
<td>Index block length.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>IXBLKCONTAINS</td>
<td>Number of keys in an index block.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DISPLACEMENT</td>
<td>Distance, in number of pages, an index entry can be stored from the referenced row.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NEXT</td>
<td>Offset to next db-key pointer (CALC key).</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>PRIOR</td>
<td>Offset to prior db-key pointer (CALC key).</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMSETS</td>
<td>For non-CALC indexes, this value is:</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>▪ 1, if the number of rows in the indexed table was greater than zero when statistics were last updated.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>▪ 0, if there were no rows in the indexed table when statistics were last updated.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>For CALC indexes, this value is the number of target pages for indexed rows when statistics were last updated.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------</td>
<td>-----------</td>
<td>-------------------</td>
</tr>
<tr>
<td>AVGMEMROWS</td>
<td>For non-CALC indexes, this value is the same as the NUMROWS column value in the SYSTEM.TABLE row for the indexed table when statistics were last updated. For CALC indexes, this value is the average number of indexed rows for each target page in the CALC index when statistics were last updated.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>LONGESTMEM</td>
<td>For non-CALC indexes, this value is the same as the NUMROWS column value in the SYSTEM.TABLE row for the indexed table when statistics were last updated. For CALC indexes, this value is the highest number of indexed rows for the same target page when statistics were last updated.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SECLONGMEM</td>
<td>For non-CALC indexes, this value is always 0. For CALC indexes, this value is the second highest number of indexed rows for the same target page when statistics were last updated.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMLONGMEM</td>
<td>For non-CALC indexes, this value is always 1. For CALC indexes, this value is the number of target pages with LONGESTMEM indexed rows targeted to the page when statistics were last updated.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>AVGMEMPAGES</td>
<td>For non-CALC indexes, this value is the same as the NUMPAGES column value in the SYSTEM.TABLE row for the indexed table when statistics were last updated. For CALC indexes, this value is the average number of distinct pages occupied by indexed rows that target to the same page when statistics were last updated. This average does not include pages accounted for in MAXMEMPAGES. The nearer this value is to 1, the greater the efficiency of the index.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
| MAXMEMPAGES     | For non-CALC indexes, this value is:  
- 0, if AVGMEMPAGES was 20 or less when statistics were last updated  
- 0, if AVGMEMPAGES was more than 20 when statistics were last updated  
For CALC indexes, this value is the number of target pages where the number of pages occupied by indexed rows that target to the same page occupied more than 20 pages when statistics were last updated. This average does not include pages accounted for in AVGMEMPAGES (above). | INTEGER   | NOT NULL           |
<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVGAMEMCLUSCNT</td>
<td>For a non-CALC index, the average number of I/Os required to read all rows of the indexed table associated with a referenced row when statistics were last updated, if one buffer page was available. This count includes I/O to read the bottom-level SR8. For a CALC index, this value is 0.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGBMEMCLUSCNT</td>
<td>For a non-CALC index, the average number of I/Os required to read all rows of the indexed table associated with a referenced row when statistics were last updated, if three buffer pages were available. This count includes I/O to read the bottom-level SR8. For a CALC index, this value is 0.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGCMEMCLUSCNT</td>
<td>For a non-CALC index, the average number of I/Os required to read all rows of the indexed table associated with a referenced row if when statistics were last updated, if five buffer pages were available. This count includes I/O to read the bottom-level SR8. For a CALC index, this value is 0.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGDMEMCLUSCNT</td>
<td>For a non-CALC index, the average number of I/Os required to read all rows of the indexed table associated with a referenced row when statistics were last updated, if 10 buffer pages were available. This count includes I/O to read the bottom-level SR8. For a CALC index, this value is 0.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>AVGEMEMCLUSCNT</td>
<td>For a non-CALC index, the average number of I/Os required to read all rows of the indexed table associated with a referenced row when statistics were last updated, if 20 buffer pages were available. This count includes I/O to read the bottom-level SR8. For a CALC index, this value is 0.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGSR8ROWS</td>
<td>For a non-CALC index, the number of SR8s in the index when statistics were last updated. For a CALC index, this value is 0.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>LONGESTSR8</td>
<td>For a non-CALC index, the number of bottom-level SR8s in the index. For a CALC index, this value is 0.</td>
<td>INTEGRER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SECLONGSR8</td>
<td>This value is always 0.</td>
<td>INTEGRER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMLONGSR8</td>
<td>For a non-CALC index, this value is 1. For a CALC index, this value is 0.</td>
<td>INTEGRER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGSR8PAGES</td>
<td>For a non-CALC index, the average number of distinct pages occupied by bottom-level SR8s for the index when statistics were last updated, if the average is 1 to 20. If the average is more than 20, the value in this column is 0. For a CALC index, this value is 0.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>MAXSR8PAGES</td>
<td>For a non-CALC index, if AVGSR8PAGES is more than 20, the value in this column is 1. Otherwise the value is 0. For a CALC index, this value is 0.</td>
<td>INTEGRER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>-----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>AVGSR8LEAFS</td>
<td>For a non-CALC index, the average number of bottom-level SR8s in the index when statistics were last updated. For a CALC index, this value is 0.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGSR8LEVELS</td>
<td>For a non-CALC index, the average highest level number in the index when statistics were last updated. Level number refers to the number of levels above the bottom level. For a CALC index, this value is 0.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGASR8CLUSCNT</td>
<td>For a non-CALC index, the average number of I/Os required to read all bottom-level SR8s in the index when statistics were last updated, if one buffer page was available. For a CALC index, this value is 0.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGBSR8CLUSCNT</td>
<td>For a non-CALC index, the average number of I/Os required to read all bottom-level SR8s in the index when statistics were last updated, if three buffer pages were available. For a CALC index, this value is 0.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGCSR8CLUSCNT</td>
<td>For a non-CALC index, the average number of I/Os required to read all bottom-level SR8s in the index when statistics were last updated, if five buffer pages were available. For a CALC index, this value is 0.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGDSR8CLUSCNT</td>
<td>For a non-CALC index, the average number of I/Os required to read all bottom-level SR8s in the index when statistics were last updated, if 10 buffer pages were available. For a CALC index, this value is 0.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVGESR8CLUSCNT</td>
<td>For a non-CALC index, the average number of I/Os required to read all bottom-level SR8s in the index when statistics were last updated, if 20 buffer pages were available. For a CALC index, this value is 0.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMUNIQKEYS</td>
<td>The number of distinct key values in the index when statistics were last updated. For a unique index, this number should match NUMROWS in the SYSTEM.TABLE row for the underlying table.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMNULLKEYS</td>
<td>Number of rows where all the index key columns in the row contained null key values when statistics were last updated.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMLONGKEYS</td>
<td>Number of distinct index key values for which all referencing rows with the same key value the indexed table occupied more than 20 pages when statistics were last updated. This situation can occur only when referencing rows can contain duplicate key values.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGDUPSPERKEY</td>
<td>For each distinct index key value, the average number of duplicate values when statistics were last updated. For a unique index, this is 1.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGPGESPERKEY</td>
<td>For each distinct index key value, the average number of pages containing rows of the indexed table with the key value when statistics were last updated. This average does not include pages accounted for in NUMLONGKEYS (above).</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>PROCKEY</td>
<td>Table procedure key: • P—If procedure key</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
### SYSTEM.INDEXKEY

**Description**

A row of SYSTEM.INDEXKEY identifies a key column defined in an index. This information is for documentation and is not used in internal processing.

View SYSCA.INDEXKEY is defined on SYSTEM.INDEXKEY.

**Note:** For more information, see SYSCA Objects.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Index name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SCHEMA</td>
<td>Schema-name qualifier for the indexed table.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TABLE</td>
<td>Name of the indexed table.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SEQUENCE</td>
<td>Internal index key sequence number.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMBER</td>
<td>Column number of the key column.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>COLUMN</td>
<td>Column name of the key column.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SORTORDER</td>
<td>Index key sort order:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ A—Ascending</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ D—Descending</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FILLER</td>
<td>Reserved for future use.</td>
<td>BINAR Y(38)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
**SYSTEM.JOURNAL**

**Description**

A row of SYSTEM.JOURNAL represents a journal file defined in the DMCL.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMCL</td>
<td>DMCL name.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NAME</td>
<td>Journal name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TYPE</td>
<td>Journal type:</td>
<td>CHAR(4)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ DISK</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ ARCH</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ TAPE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTIME</td>
<td>Date and time stamp when the journal was created.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UTIME</td>
<td>Date and time stamp when the journal was last updated.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CRITTIME</td>
<td>Date and time stamp of the last critical change made to the journal.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CUSER</td>
<td>ID of the user who created the journal.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UUSER</td>
<td>ID of the user who last updated the journal.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FILLER</td>
<td>Reserved for future use.</td>
<td>CHAR(2)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMBLOCKS</td>
<td>Number of blocks (pages) in a disk journal file.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>BLOCKSIZE</td>
<td>Archive journal block size, in bytes.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DDNAME</td>
<td>Depending on the operating system, DDName (z/OS), filename (z/VSE), or linkname of the journal file.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>-------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>ACCESSMETHOD</td>
<td>Journal file access method.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DATASPACE</td>
<td>Dataspace option:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ N—No</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ Y—Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESAREAD</td>
<td>Dataspace read:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ B—Block</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ T—Track</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ C—Cylinder</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ Blank—Dataspace no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESAPRELOAD</td>
<td>Dataspace preload.</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ N—No</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ Y—Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ Blank—Dataspace no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSNAME</td>
<td>Dataset name.</td>
<td>CHAR(44)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DISP</td>
<td>Dataset disposition.</td>
<td>CHAR(4)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>VMUSERID</td>
<td>In the z/VM environment, the user ID associated with the file.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>VMVIRTADDR</td>
<td>In the z/VM environment, the virtual address of the file.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FILLER</td>
<td>Reserved for future use.</td>
<td>BINAR(41)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
# SYSTEM.LOADHDR

## Description

A row of SYSTEM.LOADHDR represents a load module. Each row of SYSTEM.LOADHDR contains global information about a load module that resides in the DDLDCLOD or DDLCATLOD area of the dictionary.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Load module name.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>VERSION</td>
<td>Load module version number.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>RLDS</td>
<td>Number of entries in the relocation dictionary (RLD) for the load module.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>EPA</td>
<td>Entry point address (offset).</td>
<td>BINARY(4)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>LENGTH</td>
<td>Length, in bytes, of the object text for the load module.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DATE</td>
<td>Date when the load module was created (mm/dd/yy).</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TIME</td>
<td>Time when the load module was created (hhmmss).</td>
<td>CHAR(6)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TYPE</td>
<td>Flag byte for load module status and type.</td>
<td>BINARY(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------</td>
<td>-----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>SEC</td>
<td>Security class.</td>
<td>BINAR Y(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>STLENGTH</td>
<td>Reserved.</td>
<td>INTEGRER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>MODE</td>
<td>Reserved.</td>
<td>BINAR Y(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>STLEVEL</td>
<td>X'02'—Module supports SQL schema names.</td>
<td>BINAR Y(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SCHEMA</td>
<td>SQL schema name for access module.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>

**SYSTEM.ORDERKEY**

**Description**

A row of SYSTEM.ORDERKEY represents a column that is a sort key in a linked constraint. This is a column in the referencing table. This information is for documentation and is not used in internal processing.

View SYSCA.ORDERKEY is defined on SYSTEM.ORDERKEY.

**Note:** For more information, see SYSCA Objects.
### SYSTEM.SCHEMA

**Description**

A row of SYSTEM.SCHEMA represents an SQL schema.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Schema name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CTIME</td>
<td>Date and time when the schema was created.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UTIME</td>
<td>Date and time when the schema was last altered.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CUSER</td>
<td>ID of the user who created the schema.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UUSER</td>
<td>The ID of the user who last altered the schema.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TYPE</td>
<td>Type of schema:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ N—Represents non-SQL schema</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ R—SQL schema</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>-------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>SEGMENT</td>
<td>■ When TYPE is N, the segment(specified in CREATE SCHEMA) that contains the data described by the non-SQL schema. If this column is blank and TYPE is N, a segment is chosen from the database name table at runtime. ■ When TYPE is R, the segment that contains the default area. If this column and the AREA column are blank and TYPE is R, the storage area for a table associated with this schema is identified in SYSTEM.TABLE.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AREA</td>
<td>When TYPE is R, the default area for storing rows of tables associated with the schema.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NODE</td>
<td>If it was specified when the SQL schema was created, node name of the dictionary that contains the non-SQL-defined schema (when TYPE is N).</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DICTIONARY</td>
<td>If it was specified when the SQL schema was created, name of the dictionary that contains the non-SQL-defined schema (when TYPE is N).</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NTWKSCHEMA</td>
<td>Name of the non-SQL schema.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>VERSION</td>
<td>Version number of the non-SQL schema.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CHARSET</td>
<td>Default character set for all columns in the catalog.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>REFDSQLSCHEMA</td>
<td>Name of referenced SQL schema.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
**SYSTEM.SECTION**

**Description**

A row of SYSTEM.SECTION describes part or all the tree form (the form input to the optimizer) of either a search condition that defines a check constraint or a SELECT statement that defines a view. If more than one row is needed to return the entire section, the order of the section portions is represented in column SEQUENCE.

View SYSCA.SECTION is defined on SYSTEM.SECTION.

**Note:** For more information, see SYSCA Objects.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMA</td>
<td>Schema-name qualifier of the table or view.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TABLE</td>
<td>Table or view name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TYPE</td>
<td>Section type:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ C—Check constraint definition</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ V—View definition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORMAT</td>
<td>Section format; Value is I (I-tree)</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SEQUENCE</td>
<td>Sequence number of this portion of the section.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TEXT</td>
<td>Text of the section representing the check constraint or view definition.</td>
<td>BINAR Y(512)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
## SYSTEM.SEGMENT

### Description

A row of SYSTEM.SEGMENT represents a definition of a database segment.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Segment name.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CTIME</td>
<td>Date and time stamp when the segment was created.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UTIME</td>
<td>Date and time stamp when the segment was last updated.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CRITTIME</td>
<td>Date and time stamp of the last critical change made to the segment.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CUSER</td>
<td>ID of the user who created the segment.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UUSER</td>
<td>ID of the user who last updated the segment.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SCHEMA</td>
<td>Name of the SQL schema, if any, that is associated with the segment. If an SQL schema name is associated with the segment, only tables whose names are qualified by the SQL schema name can be stored in areas associated with the segment.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>PAGEGROUP</td>
<td>Identifier of the page group that contains the areas associated with the segment.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>RECSPERPAGE</td>
<td>Maximum number of rows that can be stored on a single page. The value in this column is equal to the value supplied by the user, rounded up to the nearest power of 2, minus 1.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMAREAS</td>
<td>Number of areas associated with the segment.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
### SYSTEM.SYMBOL

#### Description

SYMBOL represents a named symbol within an area whose values are used at runtime to resolve symbolic parameters named in logical definitions.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE</td>
<td>Symbolic type.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ 01—Subarea</td>
<td>SMALL INT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ 02—Symbolic displacement</td>
<td>SMALL INT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ 03—Symbolic index</td>
<td>SMALL INT</td>
<td></td>
</tr>
<tr>
<td>NAME</td>
<td>Symbolic name</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------</td>
<td>-----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>SEGMENT</td>
<td>Segment name.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AREA</td>
<td>Area name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CTIME</td>
<td>Date and time stamp when the symbolic was created.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UTIME</td>
<td>Date and time stamp when the symbolic was last updated.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CRITTIME</td>
<td>Date and time stamp of the last critical change to the symbolic.</td>
<td>TIMES TAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CUSER</td>
<td>ID of the user who created the area.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UUSER</td>
<td>ID of the user who last updated the area.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FLAG</td>
<td>Symbolic type flag; if column TYPE=:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BINAR Y(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ 01, values are:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ 'X'80'—Subarea offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ 'X'40'—VALUE1 is a percent</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ 'X'20'—VALUE2 is a percent</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ 'X'10'—Subarea space</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ 02, value is</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ 'X'80'—(Displacement)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ 03:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ 'X'80'—Index block contains</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ 'X'40'—Index size</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ 'X'20'—Index sorted key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FILLER</td>
<td>Reserved for future use.</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>VALUE1</td>
<td>Symbolic value 1.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>VALUE2</td>
<td>Symbolic value 2.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FILLER</td>
<td>Reserved for future use.</td>
<td>BINAR Y(40)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
SYSTEM.SYNTAX

Description

A row of SYSTEM.SYNTAX represents the syntax for a CREATE or ALTER TABLE statement that includes a check constraint, the syntax of a CREATE VIEW statement, or the syntax of an SQL routine. If more than one row is needed to return all the syntax, the order of the syntax portions is represented in column SEQUENCE.

View SYSCA.SYNTAX is defined on SYSTEM.SYNTAX.

Note: For more information, see SYSCA Objects.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMA</td>
<td>Schema-name qualifier of the table or view.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TABLE</td>
<td>Table or view name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TYPE</td>
<td>Syntax usage:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ C—Check constraint definition</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ V—View definition (SELECT statement)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ S—SQL routine definition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEQUENCE</td>
<td>Sequence number of this portion of the syntax.</td>
<td>SMALL INT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SYNTAX</td>
<td>The check constraint syntax or the syntax of the query expression in the view definition.</td>
<td>CHAR(80)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
**SYSTEM.TABLE**

**Description**

A row of SYSTEM.TABLE represents the definition of a table, view, function, procedure, table procedure or owner of local variables of an SQL routine, or refers to a database record in a non-SQL-defined schema referenced by a view definition (for which only SCHEMA and NAME information appears).

View SYSCA.TABLE is defined on SYSTEM.TABLE.

**Note:** For more information, see SYSCA Objects.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMA</td>
<td>Schema-name qualifier of the table or view.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the table or view.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SEGMENT</td>
<td>Segment that contains the area where table rows are stored.</td>
<td>CHAR(8)</td>
<td></td>
</tr>
<tr>
<td>AREA</td>
<td>Area where table rows are stored.</td>
<td>CHAR(18)</td>
<td></td>
</tr>
<tr>
<td>TABLEID</td>
<td>Internal record ID of the database record underlying the table.</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TYPE</td>
<td>Type of table:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>- A—View on referencing SQL schema table</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- F—Function</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- L—Owner of local variables of a routine</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- N—Record in a non-SQL-defined schema</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- P—Table procedure</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- R—Procedure</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- T—Base table</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- V—View</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------</td>
<td>-----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>LOCMODE</td>
<td>Storage location mode for a table:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ C—Clustered</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ D—Direct</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ H—CALC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ R—Row ID index</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ U—Unique calc</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Global storage allocation option for a table procedure:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ N—Non-keyed global storage</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ K—Keyed global storage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPRESS</td>
<td>Compression indicator:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ Y—Compressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ N—Uncompressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ P—Compressed with CA IDMS Presspack</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>For TYPE = R (procedure) or F (function), this is the protocol:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ I—IDMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ A—ADS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORMAT</td>
<td>Format of the database record underlying the table:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ F—Fixed length</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ V—Variable length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPDATABLE</td>
<td>When TYPE is V, updatable view indicator:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ Y—Updatable</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ N—Not updatable</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ Blank—Not known at definition time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>CHECKOPT</td>
<td>When TYPE is V, WITH CHECK OPTION indicator:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ Y—View defined with WITH CHECK OPTION</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ N—View defined without WITH CHECK OPTION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Table timestamp, used for synchronization with access module definitions,</td>
<td>TIMESTAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CTIME</td>
<td>Date and time when the table or view was created.</td>
<td>TIMESTAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UTIME</td>
<td>Date and time when the table or view was last altered.</td>
<td>TIMESTAMP</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CUSER</td>
<td>ID of the user who created the table or view.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>UUSER</td>
<td>ID of the user who last altered the table or view.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>PUTROUTINE</td>
<td>■ When TYPE = T: CA IDMS Presspack data characteristic table (DCT) name.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ When TYPE = P or R or F: external program or dialog name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GETROUTINE</td>
<td>Reserved for a table. Global storage key for a procedure.</td>
<td>CHAR(8)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>LENGTH</td>
<td>Internal length of underlying database record, including prefix.</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DATALENGTH</td>
<td>Internal length of the data portion of the underlying database record (including four-byte RDW for a compressed table).</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>-----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>PREFIXLENGTH</td>
<td>Internal length of the prefix portion of the underlying database record for a table. Length of the local work area for a procedure.</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>CTRLENGTH</td>
<td>Internal length of the control portion (without the prefix) of the underlying database record for a table. Length of the global work area for a procedure.</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FIXLENGTH</td>
<td>Internal length of the fixed portion (without the prefix) of the underlying database record.</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SECLENGTH</td>
<td>Length of the I-tree stored in the associated section table rows, for a view or check constraint.</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMSYNTAX</td>
<td>Number of rows in the syntax table for a view or check constraint.</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMCOLS</td>
<td>Number of columns in the table or view.</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMINDEXES</td>
<td>Number of indexes on the table.</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMREFERENCED</td>
<td>Number of constraints where the table is the referenced table.</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMREFERENCING</td>
<td>Number of constraints where the table is the referencing table.</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DISPLACEMENT</td>
<td>Displacement, in pages, from cluster index or constraint for a table. Estimated number of I/Os for a procedure.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>ESTROWS</td>
<td>Estimated number of rows in the table.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Column name</td>
<td>Column description</td>
<td>Data type</td>
<td>Null specification</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------</td>
<td>-----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>NUMPAGES</td>
<td>Number of pages containing rows of the table when statistics were last updated.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMROWS</td>
<td>Actual number of rows in the table when statistics were last updated.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>ROWSPERPAGE</td>
<td>Number of table rows per page when statistics were last updated.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>AVGROWLENGTH</td>
<td>Average length of a table row when statistics were last updated.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>PCTSPACEUSED</td>
<td>Percentage of space used in the area where table rows are stored when statistics were last updated.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>PCTFRAGROWS</td>
<td>Percentage of rows fragmented in storage when statistics were last updated.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMIO01</td>
<td>Number of IOs required to read all rows with 1 buffer, when statistics were last updated.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMIO03</td>
<td>Number of IOs required to read all rows with 3 buffers, when statistics were last updated.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMIO05</td>
<td>Number of IOs required to read all rows with 5 buffers, when statistics were last updated.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMIO10</td>
<td>Number of IOs required to read all rows with 10 buffers, when statistics were last updated.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>NUMIO20</td>
<td>Number of IOs required to read all rows with 20 buffers, when statistics were last updated.</td>
<td>REAL</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
| PROCMODE | Execution mode for a table procedure:  
  - U—User mode  
  - S—System mode | CHAR(1) | NOT NULL |
## SYSTEM.VIEWDEP

### Description

A row of SYSTEM.VIEWDEP identifies a table or view referenced by a view.

View SYSCA.VIEWDEP is defined on SYSTEM.VIEWDEP.

**Note:** For more information, see SYSCA Objects.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>TXNSHARING</td>
<td>Transaction sharing:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ Y—Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ N—No</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ D—Default</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROCDBNAME</td>
<td>Default Database:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ P—Current</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ not P—Null</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEFAULTINDEX</td>
<td>When TYPE is T, Row ID index indicator:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td>■ Y—Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ N—No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SECLENGTH2</td>
<td>Length of syntax for views and check constraints.</td>
<td>INTEGER</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>LANGUAGE</td>
<td>Language of an SQL-invoked routine.</td>
<td>CHAR3</td>
<td></td>
</tr>
<tr>
<td>FILLER3</td>
<td>Reserved for future use.</td>
<td>BINARY(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DYNRESULTSETS</td>
<td>Number of dynamic results sets of an SQL-invoked procedure.</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>FILLER</td>
<td>Reserved for future use.</td>
<td>BINARY(18)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>

### Column Descriptions

- **TXNSHARING**: Transaction sharing:
  - Y—Yes
  - N—No
  - D—Default

- **PROCDBNAME**: Default Database:
  - P—Current
  - not P—Null

- **DEFAULTINDEX**: When TYPE is T, Row ID index indicator:
  - Y—Yes
  - N—No

- **SECLENGTH2**: Length of syntax for views and check constraints.

- **LANGUAGE**: Language of an SQL-invoked routine.

- **FILLER3**: Reserved for future use.

- **DYNRESULTSETS**: Number of dynamic results sets of an SQL-invoked procedure.

- **FILLER**: Reserved for future use.
<table>
<thead>
<tr>
<th>Column name</th>
<th>Column description</th>
<th>Data type</th>
<th>Null specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Dependent view name.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>SEQUENCE</td>
<td>Sequence number of the table or view referenced in the dependent view.</td>
<td>SMALLINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>REFSHEMA</td>
<td>Schema-name qualifier of the table or view referenced in the dependent view.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>REFTABLE</td>
<td>Name of the table or view referenced in the dependent view.</td>
<td>CHAR(18)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>REFTYPE</td>
<td>Type of the table or view referenced in the dependent view:</td>
<td>CHAR(1)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>F—Function</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N—Record in a non-SQL-defined schema</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P—Table procedure</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R—Procedure</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T—Base table</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V—View</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FILLER</td>
<td>Reserved for future use.</td>
<td>BINAR(5)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
SYSCA Objects

SYSCA Views

SYSCA views are views defined on a subset of SYSTEM tables. They restrict information that the user can select from the SYSTEM tables to data about tables for which the user holds SELECT privilege.

If you hold SELECT privilege on the SYSCA views but not on SYSTEM tables, you can see definitions of the tables from which you are authorized to select data, but you cannot see definitions of any other tables defined in the catalog component of the dictionary.

Tables that You Can Access

SYSCA.ACCESSIBLE_TABLES is a view you can use to list the tables for which you hold a SELECT privilege. The result table of this view contains the schema name, table name, and type of table (T for base table, V for view, N for non-SQL-defined table, P for table procedure), as in this example:

```
SELECT * FROM SYSCA.ACCESSIBLE_TABLES;
*+
  *+ SCHEMA     TABLE     TYPE
  *+ ___________ __________ __________
  *+ DEMOEMPL    EMP_WORK_INFO  V
  *+ DEMOEMPL    JOB         T
  *+ DEMOPROJ    EXPERTISE    T
  *+ DEMOPROJ    PROJECT      T
  *+ DEMOPROJ    SKILL        T
  *+ EMPDEMO     INSURANCE_PLAN N
  *+ SYSCA       ACCESSIBLE_TABLES  V
  *+ SYSCA       COLUMN       V
  *+ SYSCA       SECTION      V
  *+ SYSCA       SYNTAX       V
  *+ SYSCA       TABLE        V
```

Non-SQL defined tables are visible through the SYSCA.ACCESSIBLE_TABLES view only if the dictionary name where the non-SQL-defined schema resides exactly matches the dictionary to which the SQL session is connected.
SYSCA View Names

In addition to SYSCA.ACCESSIBLE_TABLES, these SYSCA views are defined. The view name matches the name of the SYSTEM table on which it is defined, and the view column names match the SYSTEM table column names.

- SYSCA.COLUMN
- SYSCA.CONSTKEY
- SYSCA.CONSTRAINT
- SYSCA.INDEX
- SYSCA.INDEXKEY
- SYSCA.ORDERKEY
- SYSCA.SECTION
- SYSCA.SYNTAX
- SYSCA.TABLE
- SYSCA.VIEWDEP

Example

In this example, columns from SYSCA.TABLE are selected. The column values in the result table represent information about tables for which the issuing user holds SELECT privilege:

```
SET OPTIONS COMPRESS ON;
*+ Status = 0
SELECT SCHEMA, NAME, SEGMENT, AREA, TYPE, UPDATABLE
FROM SYSCA.TABLE
WHERE SCHEMA <> 'SYSCA'
ORDER BY 1, 2, 3, 4, 5;
*+
  +-----------+---------------+---------+---+-----------+-------:+
  | SCHEMA    | NAME          | SEGMENT | AREA| TYPE       | UPDATABLE|
  | DEMOEMPL   | EMP_WORK_INFO | SYSCAT  | DDLCTX | V | Y        |
  | DEMOEMPL   | JOB           | SQLDEMO | INFOAREA | T |
  | DEMOPROJ   | EXPERTISE     | PROJSEG | PROJAREA | T |
  | DEMOPROJ   | PROJECT       | PROJSEG | PROJAREA | T |
  | DEMOPROJ   | SKILL         | PROJSEG | PROJAREA | T |
```

Note: The SYSCA.TABLE view excludes tables of type N because most columns of SYSTEM.TABLE do not contain meaningful values for non-SQL-defined tables.
SYSCA Other Objects

A number of table-like objects such as views, table procedures, functions, and procedures are created in the schema SYSCA. These objects are needed by CA IDMS for miscellaneous purposes, or provide for generally useful procedures or functions.

SYSCA Pseudo Table SINGLETON_NULL

The SYSCA.SINGLETON_NULL is a pseudo table that can be used to return the results of expressions whose parameters are constants. It is defined to have one row and no columns. This table is a pseudo table because it does not exist in the catalog. However, it can be queried through a SELECT statement. Used internally by CA IDMS, it is also useful when evaluating SQL functions and other expressions with constant parameters.

Example

```sql
select USER01.TLANG1('James   ', 'Last   ')  
  from SYSCA.SINGLETON_NULL;
```

```plaintext
*+ USER_FUNC
*+  ---------
*+ James Last
```
Appendix F: Index Calculations
INDEX BLOCK CONTAINS

The following steps are used to calculate *key-count* for the INDEX BLOCK CONTAINS parameter of the CREATE INDEX statement:

1. Obtain the maximum number of index entries in an SR8 (this formula assumes 3 SR8s per page):
   - Compute the maximum size of the variable portion of an SR8:
     \[(\text{Page-size} - \text{Page-reserve} - 32)/3\] - 40 = SR8-variable-size
   - Compute the maximum number of index entries in an SR8:
     \[(\text{SR8-variable-size} / (8 + \text{Key-length})\] - 2
   
   If the resulting number of SR8 entries is less than 3, set it to 3; if greater than 8180, set it to 8180.

2. Establish the number of rows to be indexed using the greater of:
   - The estimated number of rows for the table (ESTROWS column in SYSTEM.TABLE)
   - The actual number of rows in the table (NUMROWS column in SYSTEM.TABLE)

   If both values are 0, use 1000.

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

<table>
<thead>
<tr>
<th>Number of Rows</th>
<th>Number of SR8 Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>15,625</td>
<td>25</td>
</tr>
<tr>
<td>125,000</td>
<td>50</td>
</tr>
<tr>
<td>512,000</td>
<td>80</td>
</tr>
<tr>
<td>1,000,000</td>
<td>100</td>
</tr>
<tr>
<td>2,000,376</td>
<td>126</td>
</tr>
<tr>
<td>3,375,000</td>
<td>150</td>
</tr>
<tr>
<td>5,359,375</td>
<td>175</td>
</tr>
<tr>
<td>8,000,000</td>
<td>200</td>
</tr>
<tr>
<td>15,625,000</td>
<td>250</td>
</tr>
<tr>
<td>-1</td>
<td>8180</td>
</tr>
</tbody>
</table>

4. Determine the INDEX BLOCK CONTAINS value by using the lesser of the Number of SR8 Entries from the table and the value obtained in Step 1.

   Use this value in the INDEX BLOCK CONTAINS *key-count* parameter of the definition and for IBC-key-count in calculations for DISPLACEMENT page-count.
DISPLACEMENT

Index Displacement

The following steps are used to calculate page-count for the DISPLACEMENT parameter of the CREATE INDEX statement:

1. Calculate number of bottom-level and higher-level SR8s:
   
   Set \( N = \) #-of-rows
   
   High-level-SR8s = 0
   
   Bottom-level-SR8s = 1

   Repeat
   
   \[
   N = \frac{(N + IBC-key-count - 1)}{IBC-key-count} \text{ (truncate)}
   \]

   If \( N = 1 \), exit

   If High-level-SR8s = 0,
   
   High-level-SR8s = 1
   
   Bottom-level-SR8s = \( N \)

   Else
   
   High-level-SR8s = High-level-SR8s + \( N \)

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

2. Determine the number of SR8s per page:

   - Calculate size of an SR8:
     
     \( SR8-size = 32 + (IBC-key-count + 2) \times (Key-length + 8) \)

   - Calculate number of SR8s per page:
     
     \[
     \frac{(Page-size - Page-reserve - 32)}{(SR8-size + 8)}
     \]

3. Establish the INDEX DISPLACEMENT:

   - If the number of higher-level SR8's is less than 2, set the DISPLACEMENT = High-level-SR8s. (For a one- or two-level index, displacement is 0 or 1 respectively).

   - If the number of higher-level SR8s is greater than 1, compute the displacement page count:
     
     \[
     \frac{(High-level-SR8s + SR8s-per-page - 1)}{SR8s-per-page} + 1 \text{ (truncate)}
     \]

     If the calculated displacement is greater than the number of pages in the area containing the index, then:
     
     \[
     \text{Displacement-page-count} = \frac{\text{Number-of-pages-in-area}}{2}
     \]
Table Displacement

Table displacement is the number of pages that the rows of a table are displaced from the index around which they are clustered. The table displacement is 0 if the table and index reside in different areas. It is calculated as described below if they reside in the same area:

- If the total number of SR8s is less than 2, set the table's displacement to 0.
- If the total number of SR8s is greater than 1, the table's displacement is:

\[
\frac{(\text{Total-SR8s} + \text{SR8s-per-page} - 1)}{\text{SR8s-per-page}} + 1 \text{ (truncate)}
\]

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

\[
\text{Table-displacement} = \text{Number-of-pages-in-area} \times \frac{3}{4}
\]
Appendix G: Sample COBOL Table Procedure

Sample Table Procedure Definition

The following example shows a table procedure definition.

```
create table procedure emp.org
  (top_key            unsigned numeric(4),
   level              smallint,
   mgr_id             unsigned numeric(4),
   mgr_lname          char(25),
   emp_id             unsigned numeric(4),
   emp_lname          char(25),
   start_date         DATE,
   structure_code     char(2))
external name procorgu
local work area 800
global work area 600 key emp
estimated ios 50
estimated rows 50;
create primary key org1
  on emp.org (mgr_id, start_date, emp_id)
estimated rows 5
estimated ios 5;
create key org2
  on emp.org (mgr_id)
estimated rows 5
estimated ios 5;
create key org3
  on emp.org (emp_id)
estimated rows 5
estimated ios 5;
```
Sample Table Procedure Program

The following sample program is included on the CA IDMS installation media. This program requires the employee demo database.

*RETRIEVAL
*NO-ACTIVITY-LOG
*DMLIST

IDENTIFICATION DIVISION.
PROGRAM-ID. PROCORGU.

ENVIRONMENT DIVISION.
IDMS-CONTROL SECTION.
PROTOCOL. MODE IS BATCH DEBUG
IDMS-RECORDS MANUAL.

DATA DIVISION.

SCHEMA SECTION.
DB EMPSS01 WITHIN EMPSCHM VERSION 100.

WORKING- STORAGE SECTION.
01 WORK-FIELDS.
  02 IN01-RPB PIC X(36).
  02 IN01-REQUEST.
    03 IN01-REQUEST-CODE PIC S9(8) COMP SYNC.
    03 IN01-RETURN PIC S9(8) COMP SYNC.
  02 IN01-DATE-FORMAT PIC S9(8) COMP SYNC.
  02 I PIC S9(4) COMP SYNC.
  02 ROW-FOUND-FLAG PIC X.
  88 ROW-FOUND VALUE '1'.
01 COPY IDMS SUBSCHEMA- NAMES.
01 WK-DBKEYS.
  02 WK-NEW-MGR-DBKEY PIC S9(8) COMP SYNC.
  02 WK-NEW-EMP-DBKEY PIC S9(8) COMP SYNC.
  02 WK-MGR-DBKEY PIC S9(8) COMP SYNC.
  02 WK-EMP-DBKEY PIC S9(8) COMP SYNC.
  02 WK-PRIOR-DBKEY PIC S9(8) COMP SYNC.
  02 WK-STRUCT-DBKEY PIC S9(8) COMP SYNC.
01 WK-DATE-TIME.
  02 WK-DATE.
    03 FILLER PIC 99 VALUE 19.
    03 WK-YY PIC 99.
    03 FILLER PIC X VALUE '−'.
    03 WK-MM PIC 99.
    03 FILLER PIC X VALUE '−'.
    03 WK-DD PIC 99.
  02 WK-TIME PIC X(16) VALUE '−00.00.00.000000'.

752 SQL Reference Guide
01 WK-NEW-DATE.
   02 WK-NEW-YY         PIC 99.
   02 WK-NEW-MM         PIC 99.
   02 WK-NEW-DD         PIC 99.
01 DB-MSG.
   02 FILLER             PIC X(22) VALUE
                     'Database error status '.
   02 DB-STAT             PIC X(4).
   02 FILLER             PIC X(16) VALUE ', during: '
   02 DB-VERB             PIC X(12).
01 INVDELSEQ-MSG.
   02 FILLER             PIC X(43) VALUE
                     'Internal sequence error during delete for: '
   02 DEL-PROC             PIC X(18).
01 INVUPDSEQ-MSG.
   02 FILLER             PIC X(43) VALUE
                     'Internal sequence error during update for: '
   02 UPD-PROC             PIC X(18).
01 EMPID-MSG.
   02 FILLER             PIC X(28) VALUE
                     'EMP_ID is missing or invalid'.
01 MGRID-MSG.
   02 FILLER             PIC X(28) VALUE
                     'MGR_ID is missing or invalid'.
01 STRUCTCD-MSG.
   02 FILLER             PIC X(36) VALUE
                     'STRUCTURE_CODE is missing or invalid'.

LINKAGE SECTION.

*  PROCEDURE PARAMETERS
77 TOP-KEY             PIC 9(4).
77 LEVEL-NO             PIC S9(4) COMP SYNC.
77 MGR-ID             PIC 9(4).
77 MGR-LNAME             PIC X(25).
77 EMP-ID             PIC 9(4).
77 EMP-LNAME             PIC X(25).
77 START-DATE             PIC X(10).
77 STRUCTURE-CODE         PIC XX.

*  PROCEDURE PARAMETER INDICATORS
77 TOP-KEY-I             PIC S9(4) COMP SYNC.
77 LEVEL-NO-I             PIC S9(4) COMP SYNC.
77 MGR-ID-I             PIC S9(4) COMP SYNC.
77 MGR-LNAME-I             PIC S9(4) COMP SYNC.
**Sample Table Procedure Program**

```sql
77 EMP_ID-I PIC S9(4) COMP SYNC.
77 EMP-LNAME-I PIC S9(4) COMP SYNC.
77 START-DATE-I PIC S9(4) COMP SYNC.
77 STRUCTURE-CODE-I PIC S9(4) COMP SYNC.

* CONTROL PARAMETERS
77 RESULT-IND PIC S9(4) USAGE COMP SYNC.
01 SQLSTATE.
   02 SQLSTATE-CLASS PIC XX.
   02 SQLSTATE-SUBCLASS PIC XXX.
77 PROCEDURE-NAME PIC X(18).
77 SPECIFIC-NAME PIC X(8).
77 MESSAGE-TEXT PIC X(80).
01 SQL-COMMAND-CODE PIC S9(8) USAGE COMP SYNC.
01 SQL-OP-CODE PIC S9(8) USAGE COMP SYNC.
   88 SQL-OPEN-SCAN VALUE +12.
   88 SQL-NEXT-ROW VALUE +16.
   88 SQL-CLOSE-SCAN VALUE +20.
   88 SQL-SUSPEND-SCAN VALUE +24.
   88 SQL-RESUME-SCAN VALUE +28.
   88 SQL-INSERT-ROW VALUE +32.
   88 SQL-DELETE-ROW VALUE +36.
   88 SQL-UPDATE-ROW VALUE +40.
01 INSTANCE-ID PIC S9(8) USAGE COMP SYNC.
01 LOCAL-WORK-AREA.
   02 SCAN-INFO.
      03 SCAN-MGR-DBKEY PIC S9(8) USAGE COMP SYNC.
      03 SCAN-TOP-DBKEY PIC S9(8) USAGE COMP SYNC.
      03 SCAN-STACK-STRDBKEY OCCURS 50 TIMES PIC S9(8) USAGE COMP SYNC.
      03 SCAN-STACK-EMPDBKEY OCCURS 50 TIMES PIC S9(8) USAGE COMP SYNC.
      03 SCAN-TYPE PIC S9(4) USAGE COMP SYNC.
      03 SCAN-LEVEL PIC S9(4) USAGE COMP SYNC.
      03 SCAN-MAX-LEVEL PIC S9(4) USAGE COMP SYNC.
      03 SCAN-TOPL-KEY PIC 9(4).
      03 SCAN-MGR-KEY PIC 9(4).
      03 SCAN-MGR-NAME PIC X(25).
01 GLOBAL-WORK-AREA.
   02 COPY IDMS SUBSCHEMA-CTRL.
   02 COPY IDMS RECORD EMPLOYEE.
   02 COPY IDMS RECORD STRUCTURE.
   02 RUN-UNIT-FLAG PIC X.
   88 RUN-UNIT-BOUND VALUE '1'.
```

*****************************************************************************
PROCEDURE DIVISION USING
  TOP-KEY
  LEVEL-NO
  MGR-ID
  MGR-LNAME
  EMP-ID
  EMP-LNAME
  START-DATE
  STRUCTURE-CODE
  TOP-KEY-I
  LEVEL-NO-I
  MGR-ID-I
  MGR-LNAME-I
  EMP-ID-I
  EMP-LNAME-I
  START-DATE-I
  STRUCTURE-CODE-I
  RESULT-IND
  SQLSTATE
  PROCEDURE-NAME
  SPECIFIC-NAME
  MESSAGE-TEXT
  SQL-COMMAND-CODE
  SQL-OP-CODE
  INSTANCE-ID
  LOCAL-WORK-AREA
  GLOBAL-WORK-AREA.

MAINLINE SECTION.
 *
 * PROCESS DML-ONLY OPERATIONS
 *
   IF      SQL-NEXT-ROW
       PERFORM NEXT-ROW
   ELSE IF SQL-OPEN-SCAN
       PERFORM OPEN-SCAN
   ELSE IF SQL-INSERT-ROW
       PERFORM INSERT-ROW
   ELSE IF SQL-UPDATE-ROW
       PERFORM UPDATE-ROW
   ELSE IF SQL-DELETE-ROW
       PERFORM DELETE-ROW.
   GOBACK.
DELETE-ROW SECTION.
 *
* DELETE MUST HAVE BEEN PRECEEDED BY A "NEXT ROW"
* CALL RETRIEVING THE ROW TO BE DELETED
* DELETE "CURRENT" ROW AND
*     RESET CURRENCY TO ITS PRIOR IN SET
*
MOVE SCAN-LEVEL TO I.
FIND STRUCTURE DB-KEY SCAN-STACK-STRDBKEY (I)
IF ERROR-STATUS NOT = '0000'
    PERFORM INVDELSEQ-ERROR
    GO TO DELETE-ROW-X
MOVE 'ACCEPT PRIOR' TO DB-VERB.
IF SCAN-TYPE = 3
    ACCEPT SCAN-STACK-STRDBKEY (I) FROM
        REPORTS-TO PRIOR CURRENCY
ELSE
    ACCEPT SCAN-STACK-STRDBKEY (I) FROM
        MANAGES PRIOR CURRENCY.
IF ERROR-STATUS = '0000'
    MOVE 'ERASE' TO DB-VERB
    ERASE STRUCTURE.
IF ERROR-STATUS NOT = '0000'
    PERFORM DB-ERROR.
DELETE-ROW-X.
EXIT.

INSERT-ROW SECTION.
 *
* MAKE SURE RUNUNIT IS BOUND BEFORE STORING ROW
*
PERFORM RU-BIND.
IF SQLSTATE NOT = '00000'
    GO TO INSERT-ROW-X.
PERFORM VALIDATE-INPUT.
IF SQLSTATE NOT = '00000'
    GO TO INSERT-ROW-X.
MOVE 'FIND DBKEY' TO DB-VERB.
FIND DB-KEY WK-NEW-MGR-DBKEY.
IF ERROR-STATUS = '0000'
    MOVE WK-NEW-DATE TO STRUCTURE-DATE-0460
    MOVE STRUCTURE-CODE TO STRUCTURE-CODE-0460
    MOVE 'STORE' TO DB-VERB
    STORE STRUCTURE.
IF ERROR-STATUS = '0000'
MOVE 'FIND DBKEY' TO DB-VERB
FIND DB-KEY WK-NEW-EMP-DBKEY
IF ERROR-STATUS = '0000'
   MOVE 'CONNECT' TO DB-VERB
   CONNECT STRUCTURE TO REPORTS-TO.
IF ERROR-STATUS NOT = '0000'
   PERFORM DB-ERROR.
INSERT-ROW-X.
EXIT.
OPEN-SCAN SECTION.
*
* DETERMINE TYPE OF SCAN TO DO. CHOICES:
* 1) BOM EXPLOSION BASED ON TOP KEY
* 2) DIRECT EMPLOYEES OF A GIVEN MANAGER
* 3) DIRECT MANAGERS OF A GIVEN EMPLOYEE
* 4) AREA SWEEP OF ALL MANAGERS
*
MOVE 1 TO SCAN-MAX-LEVEL.
IF MGR-ID-I = 0
   MOVE MGR-ID TO SCAN-TOP-KEY
   MOVE 2 TO SCAN-TYPE
ELSE IF EMP-ID-I = 0
   MOVE EMP-ID TO SCAN-TOP-KEY
   MOVE 3 TO SCAN-TYPE
ELSE IF TOP-KEY-I = 0
   MOVE TOP-KEY TO SCAN-TOP-KEY
   MOVE 1 TO SCAN-TYPE
   MOVE 50 TO SCAN-MAX-LEVEL
ELSE
   MOVE 4 TO SCAN-TYPE.
   MOVE -1 TO SCAN-LEVEL.
   PERFORM RU-BIND.
OPEN-SCAN-X.
EXIT.
NEXT-ROW SECTION.
*
* THE FIRST TIME THRU, SCAN-LEVEL = -1
* WE MUST POSITION OURSELVES ON THE APPROPRIATE EMPLOYEE
* ON SUBSEQUENT ENTRY, SCAN-LEVEL >= 0
*
MOVE '0' TO ROW-FOUND-FLAG.
IF SCAN-LEVEL = -1
   MOVE 0 TO SCAN-LEVEL
   PERFORM POSITION-FIRST-TIME.
IF SQLSTATE = '00000'
   PERFORM GET-FIRST-WORKER
IF SQLSTATE = '02000'
   AND SCAN-TYPE = 4
   MOVE '00000' TO SQLSTATE
PERFORM PROCESS-NEXT-MGR UNTIL SQLSTATE NOT = '00000'
OR ROW-FOUND.
* FILL IN OUTPUT VALUES IF SUCCESSFULLY RETRIEVED ROW
*
IF ROW-FOUND
   MOVE SCAN-LEVEL TO LEVEL-NO
   IF SCAN-TYPE = 3
      MOVE SCAN-MGR-KEY TO EMP-ID
      MOVE SCAN-MGR-NAME TO EMP-LNAME
      MOVE EMP-ID-0415 TO MGR-ID
      MOVE EMP-LAST-NAME-0415 TO MGR-LNAME
   ELSE
      MOVE SCAN-MGR-KEY TO MGR-ID
      MOVE SCAN-MGR-NAME TO MGR-LNAME
      MOVE EMP-ID-0415 TO EMP-ID
      MOVE EMP-LAST-NAME-0415 TO EMP-LNAME
   END-IF
   MOVE STRUCTURE-YEAR-0460 TO WK-YY
   MOVE STRUCTURE-MONTH-0460 TO WK-MM
   MOVE STRUCTURE-DAY-0460 TO WK-DD
   MOVE 5 TO IN01-REQUEST-CODE
   MOVE 2 TO IN01-DATE-FORMAT
   CALL 'IDMSIN01' USING IN01-RPB
   IN01-REQUEST
   IN01-DATE-FORMAT
   WK-DATE-TIME
   START-DATE
   MOVE STRUCTURE-CODE-0460 TO STRUCTURE-CODE
   MOVE 0 TO LEVEL-NO-I
   MOVE 0 TO MGR-ID-I
   MOVE 0 TO MGR-LNAME-I
   MOVE 0 TO EMP-ID-I
   MOVE 0 TO EMP-LNAME-I
   MOVE 0 TO START-DATE-I
   MOVE 0 TO STRUCTURE-CODE-I
   IF SCAN-TYPE = 1
      MOVE 0 TO TOP-KEY-I
      MOVE SCAN-TOP-KEY TO TOP-KEY
   ELSE
      MOVE -1 TO TOP-KEY-I.
   END-IF
   NEXT-ROW-X.
   EXIT.
   UPDATE-ROW SECTION.
   *
   * UPDATE MUST HAVE BEEN PRECEDED BY A "NEXT ROW"
   * CALL RETRIEVING THE ROW TO BE UPDATED
   * UPDATE "CURRENT" ROW
   * IF CHANGING OWNERS (MANAGER OR EMPLOYEE)
   * ADJUST SET CONNECTIONS APPROPRIATELY
   *
   PERFORM VALIDATE-INPUT.
IF SQLSTATE NOT = '00000'
  GO TO UPDATE-ROW-X.
MOVE SCAN-LEVEL TO I.
OBTAIN STRUCTURE DB-KEY SCAN-STACK-STRDBKEY (I)
IF ERROR-STATUS NOT = '0000'
  PERFORM INVUPDSEQ-ERROR
  GO TO UPDATE-ROW-X.
MOVE 'ACCEPT OWNR' TO DB-VERB.
ACCEPT WK-MGR-DBKEY FROM MANAGES OWNER CURRENCY.
IF ERROR-STATUS = '0000'
  ACCEPT WK-EMP-DBKEY FROM REPORTS TO OWNER CURRENCY.
IF ERROR-STATUS = '0000'
  AND WK-MGR-DBKEY NOT = WK-NEW-MGR-DBKEY
    PERFORM SWITCH-MANAGERS.
  IF ERROR-STATUS = '0000'
    AND SQLSTATE = '00000'
    AND WK-EMP-DBKEY NOT = WK-NEW-EMP-DBKEY
      PERFORM SWITCH-EMPLOYEES.
  IF ERROR-STATUS = '0000'
    AND SQLSTATE = '00000'
    AND STRUCTURE-CODE NOT = STRUCTURE-CODE-0460
    OR WK-NEW-DATE NOT = STRUCTURE-DATE-0460
      MOVE WK-NEW-DATE TO STRUCTURE-DATE-0460
      MOVE STRUCTURE-CODE TO STRUCTURE-CODE-0460
      MOVE 'MODIFY' TO DB-VERB
      MODIFY STRUCTURE.
      IF ERROR-STATUS NOT = '0000'
        AND SQLSTATE = '00000'
          PERFORM DB-ERROR.
UPDATE-ROW-X.
EXIT.
Appendix G: Sample COBOL Table Procedure  761
IF ERROR-STATUS = '0000'
AND SCAN-LEVEL < SCAN-MAX-LEVEL
AND I = 0
    PERFORM PUSH-STACK
    PERFORM GET-WORKER-INFO
ELSE
    IF I = 0
        AND ERROR-STATUS NOT = '0307'
        AND ERROR-STATUS NOT = '0000'
        PERFORM DB-ERROR
    ELSE
        PERFORM GET-NEXT-ROW UNTIL SQLSTATE NOT = '0000'
    OR ROW-FOUND.
GET-NEXT-ROW SECTION.
*
* IF THE STACK IS EMPTY, WE'VE PROCESSED ALL THE ROWS
* OTHERWISE REPOSITION ON THE RECORD WHOSE DBKEY IS AT THE
* TOP OF THE STACK AND OBTAIN THE NEXT IN SET.
* IF END-OF-SET IS ENCOUNTERED,
* WE'VE PROCESSED ALL THE WORKERS AT THIS
* LEVEL AND WE MUST MOVE UP A LEVEL TO CONTINUE
*
* IF SCAN-LEVEL = 0
    MOVE '02000' TO SQLSTATE
ELSE
    MOVE 'FIND DBKEY' TO DB-VERB
    MOVE SCAN-LEVEL TO I
    FIND DB-KEY SCAN-STACK-STRDBKEY (I)
    IF ERROR-STATUS = '0000'
        MOVE 'OBTAIN NEXT' TO DB-VERB
        IF SCAN-TYPE = 3
            OBTAIN NEXT STRUCTURE WITHIN REPORTS-TO
        ELSE
            OBTAIN NEXT STRUCTURE WITHIN MANAGES
        END-IF
    END-IF
    IF ERROR-STATUS = '0000'
        PERFORM GET-WORKER-INFO
    ELSE IF ERROR-STATUS = '0307'
        PERFORM POP-STACK
    ELSE
        PERFORM DB-ERROR.
GET-WORKER-INFO SECTION.
*
* SAVE THE CURRENT DBKEY ON THE STACK
* RETRIEVE THE NAME OF THE CURRENT WORKER
* 
    MOVE SCAN-LEVEL TO I.
    MOVE DBKEY TO SCAN-STACK-STRDBKEY (I).
MOVE 'OBTAIN OWNER' TO DB-VERB.
   IF SCAN-TYPE = 3
      OBTAIN OWNER WITHIN MANAGES
   ELSE
      OBTAIN OWNER WITHIN REPORTS-TO.
   IF ERROR-STATUS = '0000'
      MOVE DBKEY TO SCAN-MGR-DBKEY
      MOVE DBKEY TO SCAN-STACK-EMPDBKEY (I)
      MOVE '1' TO ROW-FOUND-FLAG
   ELSE
      PERFORM DB-ERROR.
   END-IF
   END-IF
   IF ERROR-STATUS = '0000'
      MOVE EMP-ID-0415 TO SCAN-STACK-EMPDBKEY (I)
      END-IF
   IF SCAN-TYPE = 1
      OR SCAN-TYPE = 2
      OR SCAN-TYPE = 3
      MOVE SCAN-STACK-STRDBKEY (I)
      OBTAIN CALC EMPLOYEE
   ELSE
      OBTAIN FIRST EMPLOYEE WITHIN EMP-DEMO-REGION
      IF ERROR-STATUS = '0000'
         MOVE EMP-ID-0415 TO SCAN-TOP-KEY.
      END-IF
      IF ERROR-STATUS = '0000'
      MOVE DBKEY TO SCAN-MGR-DBKEY
MOVE DBKEY TO SCAN-TOPT-DBKEY
ELSE
    MOVE '02000' TO SQLSTATE.
PROCESS-NEXT-MGR SECTION.
*
* CONTINUE WITH AREA SWEEP ON MANAGERS...
* FIND NEXT EMPLOYEE IN AREA AND RETURN ALL THEIR
* DIRECT EMPLOYEES.
* ON EXIT, SQLSTATE = '02000' IF LAST EMPLOYEE PROCESSED
* '00000' IF MORE EMPLOYEES TO PROCESS
* '38XXX' IF ERROR
*
FIND EMPLOYEE DB-KEY SCAN-TOPT-DBKEY.
IF ERROR-STATUS = '0000'
    OBTAIN NEXT EMPLOYEE WITHIN EMP-DEMO-REGION.
IF ERROR-STATUS = '0000'
    MOVE EMP-ID-0415 TO SCAN-TOPT-KEY
    MOVE DBKEY TO SCAN-MGR-DBKEY
    MOVE DBKEY TO SCAN-TOPT-DBKEY
    PERFORM GET-FIRST-WORKER
    IF SQLSTATE = '02000'
        MOVE '00000' TO SQLSTATE
    END-IF
ELSE
    MOVE '02000' TO SQLSTATE.
PUSH-STACK SECTION.
*
* WHEN WE PUSH THE STACK, WE ALSO HAVE A NEW MANAGER
*
    MOVE EMP-ID-0415 TO SCAN-MGR-KEY
    MOVE EMP-LAST-NAME-0415 TO SCAN-MGR-NAME
    ADD 1 TO SCAN-LEVEL.

RU-BIND SECTION.
*
* BIND RUNUNIT AND READY AREA...
* IF RUNUNIT ALREADY BOUND, IGNORE. IT JUST MEANS
* ANOTHER SCAN HAD CAUSED IT TO BE BOUND PREVIOUSLY.
*
    IF RUN-UNIT-BOUND
        GO TO RU-BINDX.
    MOVE 'BIND RUNUNIT' TO DB-VERB
    BIND RUN-UNIT DBNAME 'EMPDEMO'.
    IF ERROR-STATUS = '0000'
        MOVE '1' TO RUN-UNIT-FLAG
    BIND EMPLOYEE
    BIND STRUCTURE
    MOVE 'READY AREA' TO DB-VERB
READY EMP-DEMO-REGION USAGE-MODE UPDATE
ELSE
  IF ERROR-STATUS = '1477'
  OR ERROR-STATUS = '0077'
    MOVE '0000' TO ERROR-STATUS.
  IF ERROR-STATUS NOT = '0000'
    PERFORM DB-ERROR.
RU-BINDX. EXIT.

SWITCH-EMPLOYEES SECTION.
  IF SCAN-TYPE = 3
    MOVE 'ACCEPT Prio' TO DB-VERB
    ACCEPT SCAN-STACK-STRDBKEY (I) FROM
    REPORTS-TO PRIOR CURRENCY.
    MOVE DBKEY TO WK-STRUCT-DBKEY.
    IF ERROR-STATUS = '0000'
      MOVE 'DISCONNECT' TO DB-VERB
      DISCONNECT STRUCTURE FROM REPORTS-TO.
    IF ERROR-STATUS = '0000'
      FIND DB-KEY WK-NEW-EMP-DBKEY
      FIND DB-KEY WK-STRUCT-DBKEY
      MOVE 'CONNECT' TO DB-VERB
      CONNECT STRUCTURE TO REPORTS-TO.
    IF ERROR-STATUS NOT = '0000'
      PERFORM DB-ERROR.

SWITCH-MANAGERS SECTION.
  MOVE 'ACCEPT Prio' TO DB-VERB
  ACCEPT WK-PRIOR-DBKEY FROM
  REPORTS-TO PRIOR CURRENCY.
  IF SCAN-TYPE NOT = 3
    AND ERROR-STATUS = '0000'
    ACCEPT SCAN-STACK-STRDBKEY (I) FROM
    MANAGES PRIOR CURRENCY.
  IF ERROR-STATUS = '0000'
    MOVE 'ERASE' TO DB-VERB
    ERASE STRUCTURE.
  IF ERROR-STATUS = '0000'
    FIND DB-KEY WK-NEW-MGR-DBKEY
    MOVE 'STORE' TO DB-VERB
    STORE STRUCTURE
    MOVE DBKEY TO WK-STRUCT-DBKEY.
  IF ERROR-STATUS = '0000'
    FIND DB-KEY WK-PRIOR-DBKEY
    FIND DB-KEY WK-STRUCT-DBKEY
    MOVE 'CONNECT' TO DB-VERB
    CONNECT STRUCTURE TO REPORTS-TO.
  IF ERROR-STATUS NOT = '0000'
    PERFORM DB-ERROR.
VALIDATE-INPUT SECTION.

* VALIDATE EMPLOYEE-ID
  IF EMP-ID-I = 0
    MOVE EMP-ID TO EMP-ID-0415
    OBTAIN CALC EMPLOYEE.
  IF ERROR-STATUS = '0326'
    OR EMP-ID-I NOT = 0
    PERFORM EMPI-ERROR
    GO TO VALIDATE-X.
  IF ERROR-STATUS NOT = '0000'
    MOVE 'OBTAIN CALC' TO DB-VERB
    PERFORM DB-ERROR
    GO TO VALIDATE-X.
  MOVE DBKEY TO WK-NEW-EMP-DBKEY.

* VALIDATE MANAGER-ID
  IF MGR-ID-I = 0
    MOVE MGR-ID TO EMP-ID-0415
    OBTAIN CALC EMPLOYEE.
  IF ERROR-STATUS = '0326'
    OR MGR-ID-I NOT = 0
    PERFORM MGRID-ERROR
    GO TO VALIDATE-X.
  IF ERROR-STATUS NOT = '0000'
    MOVE 'OBTAIN CALC' TO DB-VERB
    PERFORM DB-ERROR
    GO TO VALIDATE-X.
  MOVE DBKEY TO WK-NEW-MGR-DBKEY.

* VALIDATE STRUCTURE-CODE & DATE
  MOVE STRUCTURE-CODE TO STRUCTURE-CODE-0460.
  IF (ADMIN-0460
    OR PROJECT-0460)
    AND STRUCTURE-CODE-I = 0
    NEXT SENTENCE
  ELSE
    PERFORM STRUCTCD-ERROR.
  IF START-DATE-I = 0
    MOVE 5 TO IN01-REQUEST-CODE
    MOVE 0 TO IN01-DAY-REQUEST-CODE
    CALL 'IDMSIN01' USING IN01-REQUEST
    IN01-DAY-REQUEST-CODE
    START-DATE
    WK-DATE-TIME
    MOVE WK-YY TO WK-NEW-YY
    MOVE WK-DD TO WK-NEW-DD
    MOVE WK-MM TO WK-NEW-MM
  ELSE
    ACCEPT WK-NEW-DATE FROM DATE.
  VALIDATE-X.
EXIT.

***********************************************************************
****        ERROR ROUTINES                                 ****
***********************************************************************

DB-ERROR SECTION.
    MOVE '38001' TO SQLSTATE.
    MOVE ERROR-STATUS TO DB-STAT.
    MOVE DB-MSG TO MESSAGE-TEXT.

INVDELSEQ-ERROR SECTION.
    MOVE '38006' TO SQLSTATE.
    MOVE PROCEDURE-NNAME TO DEL-PROC.
    MOVE INVDELSEQ-MSG TO MESSAGE-TEXT.

INVUPDSEQ-ERROR SECTION.
    MOVE '38007' TO SQLSTATE.
    MOVE PROCEDURE-NNAME TO UPD-PROC.
    MOVE INVUPDSEQ-MSG TO MESSAGE-TEXT.

EMPID-ERROR SECTION.
    MOVE '38008' TO SQLSTATE.
    MOVE EMPID-MSG TO MESSAGE-TEXT.

MGRID-ERROR SECTION.
    MOVE '38009' TO SQLSTATE.
    MOVE MGRID-MSG TO MESSAGE-TEXT.

STRUCTCD-ERROR SECTION.
    MOVE '38010' TO SQLSTATE.
    MOVE STRUCTCD-MSG TO MESSAGE-TEXT.
Appendix H: DISPLAY and PUNCH Syntax

DISPLAY and PUNCH Syntax

The following entity type options can be specified:

- ACCESS MODULE
- CALC KEY
- CONSTRAINT
- FUNCTION
- INDEX
- KEY
- PROCEDURE
- SCHEMA
- TABLE
- TABLE PROCEDURE
- VIEW

DISPLAY/PUNCH ALL syntax for SQL DDL entities is presented first, followed by DISPLAY/PUNCH statements for each entity type.
DISPLAY and PUNCH Operations

The DISPLAY and PUNCH operations produce as output the SQL statements that describe the named entity. DISPLAY and PUNCH operations do not update the entity description. You can choose to display or punch all the entity occurrences defined within an entity or only specific entity occurrences.

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

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

**Benefit**

With the DISPLAY and PUNCH support for SQL DDL entities, you can easily display or punch entity definitions and change them, or migrate their definitions from one environment to another. For example, you can migrate definitions from one schema to another in the same catalog, and from one catalog to another in the same or different Central Version.

DISPLAY/PUNCH ALL Statement

The DISPLAY/PUNCH ALL statement displays all occurrences of an entity type. The basic syntax for each entity type is the same. The entity-option keywords vary by entity type and are presented in a table in "Usage" later in this section.

**Syntax**

```
DISPLAY/PUNCH ALL entity-type
  | entity-count
  | FIRST/LAST
  | entity
  | WHERE conditional-expression
  | AS COMments
  | SYNTAX

VERB
  | DISPLAY
  | ALTER
  | CREATE
  | DROP
  | PUNCH
```
**Parameters**

**ALL**

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

**Note:** For online users: with many entity occurrences, ALL may slow response time.

**FIRst**

Lists the first occurrence of the named entity type.

**LAST**
DISPLAY/PUNCH ALL Statement

Lists the last occurrence of the named entity type.

**entity-count**

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

**entity-type**

Identifies the entity type that is the object of the DISPLAY/PUNCH ALL request. Valid values appear in the table under "Usage" in this section.

**WHERE conditional-expression**

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

The outcome of a test for the condition determines which occurrences of the named entity type are selected for display.

**mask-comparison**

Compares an entity type operand with a mask value.

**entity-option-keyword**

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

**CONTAINS**

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

**MATCHES**

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

**'mask-value'**

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

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>Matches any alphabetic character in <em>entity-option-keyword</em>.</td>
</tr>
<tr>
<td>#</td>
<td>Matches any numeric character in <em>entity-option-keyword</em>.</td>
</tr>
<tr>
<td>*</td>
<td>Matches any character in <em>entity-option-keyword</em>.</td>
</tr>
</tbody>
</table>
value-comparison

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

'character-string-literal'

Identifies a character string enclosed in quotes.

numeric-literal

Identifies a numeric value.

entity-option-keyword

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

IS

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

NE

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

EQ/=  

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

GT/>  

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

LT/<  

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

GE

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

LE

 Specifies that the left operand must be less than or equal to the right operand for the condition to be true.
NOT

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

AND

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

OR

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

AS COMMENTS

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

AS SYNTAX

Outputs access module syntax which can be edited and resubmitted to the command facility.

VERB DISPLAY/ALTER/CREATE/DROP/PUNCH

Specifies the verb with which the entity statement is to be displayed or punched. For example, if VERB CREATE is specified, the output of the DISPLAY/PUNCH statement is a CREATE statement; if VERB ALTER is specified, the output is an ALTER statement; and so on. The default is VERB DISPLAY.
Usage

*Output Contains only enough Information to DISPLAY/PUNCH Entity*

Output produced by DISPLAY or PUNCH ALL consists only of the information necessary to execute a DISPLAY/PUNCH request for each entity occurrence.

*Valid Entity Option Keywords for Conditional Expressions*

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

<table>
<thead>
<tr>
<th>Entity type</th>
<th>Entity-option keyword</th>
<th>Selects based on</th>
</tr>
</thead>
<tbody>
<tr>
<td>All entity types</td>
<td>entity-type NAME</td>
<td>Unqualified name &amp;sub1.</td>
</tr>
<tr>
<td></td>
<td>entity-type</td>
<td>Unqualified name &amp;sub1.</td>
</tr>
<tr>
<td></td>
<td>FULL entity-type NAME</td>
<td>Qualified name &amp;sub1.</td>
</tr>
<tr>
<td></td>
<td>DATE CREATED</td>
<td>Date (MM/DD/YY) occurrence</td>
</tr>
<tr>
<td></td>
<td>MONth CREATED</td>
<td>Month occurrence created</td>
</tr>
<tr>
<td></td>
<td>DAY CREATED</td>
<td>Day occurrence created</td>
</tr>
<tr>
<td></td>
<td>YEAR CREATED</td>
<td>Year occurrence created</td>
</tr>
<tr>
<td></td>
<td>DATE Last UPDATED</td>
<td>Date (MM/DD/YY) occurrence</td>
</tr>
<tr>
<td></td>
<td>MONth last UPDATED</td>
<td>Month occurrence last updated**</td>
</tr>
<tr>
<td></td>
<td>DAY last UPDATED</td>
<td>Month occurrence last updated**</td>
</tr>
<tr>
<td></td>
<td>YEAR last UPDATED</td>
<td>Day occurrence last updated**</td>
</tr>
<tr>
<td></td>
<td>CREated by</td>
<td>Year occurrence last updated**</td>
</tr>
<tr>
<td></td>
<td>PREpared by</td>
<td>User who created occurrence**</td>
</tr>
<tr>
<td></td>
<td>REVised by</td>
<td>User who created occurrence**</td>
</tr>
<tr>
<td></td>
<td>LAST UPDateD by</td>
<td>User who created occurrence**</td>
</tr>
<tr>
<td></td>
<td></td>
<td>User who last updated occurrence**</td>
</tr>
<tr>
<td></td>
<td></td>
<td>User who last updated occurrence**</td>
</tr>
</tbody>
</table>
### DISPLAY/PUNCH ALL Statement

<table>
<thead>
<tr>
<th>Entity type</th>
<th>Entity-option keyword</th>
<th>Selects based on</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS MODULE</td>
<td>AM name</td>
<td>Unqualified access module name &amp;sub1.</td>
</tr>
<tr>
<td></td>
<td>SCHema name</td>
<td>Name of access module's schema</td>
</tr>
<tr>
<td></td>
<td>Version</td>
<td>Version number</td>
</tr>
<tr>
<td></td>
<td>FULL TABle NAME</td>
<td>Qualified name of a table referenced by the access module</td>
</tr>
<tr>
<td></td>
<td>TABle SCHema name</td>
<td>Schema name of a table referenced by the access module</td>
</tr>
<tr>
<td></td>
<td>TABle name</td>
<td>Unqualified name of a table referenced by the access module</td>
</tr>
<tr>
<td></td>
<td>DATe COMpiled</td>
<td>Date (MM/DD/YY) access module compiled</td>
</tr>
<tr>
<td></td>
<td>COMpiled</td>
<td>Date (MM/DD/YY) access module compiled</td>
</tr>
<tr>
<td></td>
<td>MONth COMpiled</td>
<td>Month access module compiled</td>
</tr>
<tr>
<td></td>
<td>DAY COMpiled</td>
<td>Day access module compiled</td>
</tr>
<tr>
<td></td>
<td>YEAr COMpiled</td>
<td>Year access module compiled</td>
</tr>
<tr>
<td>CALC KEY</td>
<td>SCHema name</td>
<td>Schema name of the table containing the CALC key</td>
</tr>
<tr>
<td></td>
<td>TABle name</td>
<td>Unqualified name of table containing the CALC key</td>
</tr>
<tr>
<td>Entity type</td>
<td>Entity-option keyword</td>
<td>Selects based on</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>CONSTRAINT</td>
<td>SCHema name</td>
<td>Schema name of the constraint</td>
</tr>
<tr>
<td></td>
<td>REFERENCEDed FULLY TAble NAME</td>
<td>Qualified name of the referenced table</td>
</tr>
<tr>
<td></td>
<td>REFERENCEDed table</td>
<td>Schema name of the referenced table</td>
</tr>
<tr>
<td></td>
<td>SCHema name</td>
<td>Unqualified name of the referenced table</td>
</tr>
<tr>
<td></td>
<td>REFERENCEDed TABLE name</td>
<td>Unqualified name of the referenced table</td>
</tr>
<tr>
<td></td>
<td>REFERENCING FULLY TABLE NAME</td>
<td>Qualified name of the referencing table</td>
</tr>
<tr>
<td></td>
<td>REFERENCING table</td>
<td>Schema name of the referencing table</td>
</tr>
<tr>
<td></td>
<td>SCHema name</td>
<td>Unqualified name of the referencing table</td>
</tr>
<tr>
<td></td>
<td>REFERENCING TABLE name</td>
<td>Unqualified name of the referencing table</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>SCHema name</th>
<th>Schema name of the function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EXTernal NAME</td>
<td>Name of the program or dialog to process the function</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INDEX</th>
<th>SCHema name</th>
<th>Schema name of the indexed table</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TABLE name</td>
<td>Unqualified name of the indexed table</td>
</tr>
<tr>
<td></td>
<td>FULLY AREa NAME</td>
<td>Qualified name of the area containing the index</td>
</tr>
<tr>
<td></td>
<td>SEGment name</td>
<td>Segment name of the area containing the index</td>
</tr>
<tr>
<td></td>
<td>AREa name</td>
<td>Unqualified name of the area containing the index</td>
</tr>
<tr>
<td>Entity type</td>
<td>Entity-option keyword</td>
<td>Selects based on</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>KEY</td>
<td>SCHEMA name</td>
<td>Schema name of the keyed table procedure</td>
</tr>
<tr>
<td></td>
<td>TABLE procedure name</td>
<td>Unqualified name of the keyed table procedure</td>
</tr>
<tr>
<td></td>
<td>FULL TABLE</td>
<td>Qualified name of the keyed table procedure</td>
</tr>
<tr>
<td></td>
<td>PROCEDURE name</td>
<td>Schema name of the procedure</td>
</tr>
<tr>
<td></td>
<td>EXTERNAL NAME</td>
<td>Name of the program or dialog called to process the procedure</td>
</tr>
<tr>
<td>SCHEMA</td>
<td>Type</td>
<td>Type of Schema (NONSQL or SQL)</td>
</tr>
<tr>
<td></td>
<td>full DICTname</td>
<td>NonSQL Schema Dictionary name</td>
</tr>
<tr>
<td></td>
<td>DBName</td>
<td>NonSQL Schema DBName</td>
</tr>
<tr>
<td></td>
<td>NODE name</td>
<td>NonSQL Schema Node Name</td>
</tr>
<tr>
<td></td>
<td>NODE name</td>
<td>NonSQL Schema Node Name</td>
</tr>
<tr>
<td></td>
<td>NONSQL SCHEMA</td>
<td>Name of the NonSQL Schema</td>
</tr>
<tr>
<td></td>
<td>nonsql schema version</td>
<td>Version of the NonSQL Schema</td>
</tr>
<tr>
<td></td>
<td>DEFAULT FULL AREA NAME</td>
<td>Qualified area name of the Schema’s default area</td>
</tr>
<tr>
<td></td>
<td>default SEGMENT name</td>
<td>Schema’s default area Segment name of</td>
</tr>
<tr>
<td></td>
<td>default AREA name</td>
<td>Schema’s default area Unqualified area name of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Schema’s default area</td>
</tr>
</tbody>
</table>
### Entity type — Entity-option keyword — Selects based on

<table>
<thead>
<tr>
<th>Entity type</th>
<th>Entity-option keyword</th>
<th>Selects based on</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE</td>
<td>SCHEMA name</td>
<td>Schema name of the table</td>
</tr>
<tr>
<td></td>
<td>FULL AREA NAME</td>
<td>Qualified area name containing the table</td>
</tr>
<tr>
<td></td>
<td>SEGMENT name</td>
<td>Segment name of the area containing the table</td>
</tr>
<tr>
<td></td>
<td>AREA name</td>
<td>Unqualified name of the area containing the table</td>
</tr>
<tr>
<td>TABLE PROCEDURE</td>
<td>SCHEMA name</td>
<td>Schema name of the table procedure</td>
</tr>
<tr>
<td></td>
<td>EXTERNAL NAME</td>
<td>Name of the program called to process the table procedure</td>
</tr>
</tbody>
</table>

**Note:** Unqualified name selections are based on the primary name of the entity occurrence only. To select based on the fully qualified occurrence name, token FULL NAME must be specified. SQL components with qualified names are specified in the table below.

**You can specify this keyword option only when using SCHEMA, TABLE, FUNCTION, PROCEDURE, TABLE PROCEDURE, and VIEW entities.**
Fully Qualified Names of SQL Components

The fully qualified names of SQL components are listed in the following table:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Fully qualified name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS MODULE</td>
<td>schema-name.access-module-name</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>schema-name.function-name</td>
</tr>
<tr>
<td>PROCEDURE</td>
<td>schema-name.procedure-name</td>
</tr>
<tr>
<td>TABLE</td>
<td>schema-name.table-name</td>
</tr>
<tr>
<td>TABLE PROCEDURE</td>
<td>schema-name.table-procedure-name</td>
</tr>
<tr>
<td>VIEW</td>
<td>schema-name.view-name</td>
</tr>
</tbody>
</table>

Date and Year 2000 Support in DISPLAY/PUNCH Statements

You can use date selection criteria as well as year 2000 support in DISPLAY ALL statements to display SQL entities.

You implement date selection criteria in these WHERE clause options:

- DATE CREATED
- DATE LAST UPDATED

You can specify the date as a value-comparison string in the form 'MM/DD/YY' in the right-hand side of the conditional expression. CA IDMS extracts it in CCMMDDYY form to accurately determine the relationship of dates. The following DISPLAY ALL statement example establishes a search criteria to identify the schemas whose DATE CREATED values are greater than the specified string.

DISPLAY ALL SCHEMAS WHERE DATE CREATED > '01/01/96';

The DISPLAY ALL process determines that the date '01/01/96' is greater than the date '12/31/95'.

Alternatively, you may specify the value-comparison string on either side of the conditional expression in the form 'CCYMMDD' to achieve the same results.
You can also substitute day, month, or year for each of these WHERE clause options. For example, this DISPLAY ALL statement specifies a search condition that is based on month and year:

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

**Default Order of Precedence Applied to Logical Operators**

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

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

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

**Example**

The following example displays all ACCESS MODULES compiled since June 1, 1995:

```
DISPLAY ALL ACCESS MODULES
  WHERE DATE CREATED GT '06/01/95'
AS SYNTAX.
```

**DISPLAY/PUNCH ACCESS MODULE**

The ACCESS MODULE, DISPLAY/PUNCH statement displays or punches an access module.

**Authorization**

To issue a DISPLAY/PUNCH ACCESS MODULE statement, you must have the DISPLAY privilege on the requested access module.
## Syntax

```
DISPLAY/PUNCH ACCESS MODULE is access-module-name

Version version-number

WITH ALL AS Comments
  | ALSo WITH NONE
  | WITHOUT DETails
  | ALSo WITHOUT HISTORY TABLEs

VERb CREATE
  | ALTER
  | DROP
```

## Parameters

- **schema-name.**
  Specifies the schema for the access module. *Schema-name* must identify the schema associated with the version of the access module being modified. If you do not specify *schema-name*, the value used by the command facility is the current schema for your SQL session.

- **access-module-name**
  Specifies the name of the access module to display or punch. *Access-module-name* must identify an access module defined and stored in the dictionary.

- **Version is version-number**
  Specifies the version number of the access module. *Version-number* is a unique integer in the range 1 through 9999. 1 is the default.

- **HIGHest**
  Specifies the highest version number associated with the access module.

- **LOWest**
  Specifies the lowest version number associated with the access module.
WITH
Lists the requested information, in addition to the information that is always included, such as the entity occurrence name.

WITHOUT
Does not list the specified options. Other options in effect through the WITH or ALSO WITH clauses in the current DISPLAY statement are displayed.

ALSO WITH
Lists the requested information, in addition to the information requested in previously issued DISPLAY WITH and DISPLAY ALSO WITH statements for the named entity.

ALSO WITHOUT
Does not list the specified options.

ALL
Specifies the display of all the information associated with the requested entity occurrence.

NONE
Specifies the display of the name of the requested entity occurrence. NONE is meaningful only when the WITH clause is specified.

DETAILS
Specifies the display of entity-specific descriptions.

HISTORY
Specifies the display of the date the access module was compiled.

TABLES
Specifies the display of all tables associated with the requested access module.

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

AS SYNTAX
Outputs access module syntax which can be edited and resubmitted to the command facility.

VERB CREATE/ALTER/DROP
Specifies the verb with which the entity statement is to be displayed or punched. For example, if VERB CREATE is specified, the output of the DISPLAY/PUNCH statement is a CREATE statement; if VERB ALTER is specified, the output is an ALTER statement; and so on. The default is VERB CREATE.
DISPLAY/PUNCH CALC KEY

The DISPLAY/PUNCH CALC KEY statement displays or punches a CALC key definition in the dictionary.

Authorization

To issue a DISPLAY/PUNCH CALC KEY statement, you must either own or have the ALTER privilege on the table on which the CALC key is defined.

Syntax

```
DISPLAY/PUNCH CALC KEY statement:
   DISPLAY | PUNCH CALC key ON table-name
   WITH     | WITHOUT | ALL | NONE | AS
   ALSO WITH| DETAILS | HISTORY
   VERb     | CREATE | DROP | PUNCH
```
Parameters

**schema-name.**

Identifies the schema associated with the named table.

If you do not specify `schema-name`, it defaults to the current schema associated with your SQL session, if the statement is entered through the command facility or executed dynamically.

**table-name**

Specifies the name of the table on which the CALC key is defined. `Table-name` must be the name of a table defined in the dictionary.

**WITH**

Lists the requested information, in addition to the information that is always included, such as the entity occurrence name.

**WITHOut**

Does not list the specified options. Other options in effect through the WITH or ALSO WITH clauses in the current DISPLAY statement are displayed.

**ALSo WITh**

Lists the requested information, in addition to the information requested in previously issued DISPLAY WITH and DISPLAY ALSO WITH statements for the named entity.

**ALSo WITHOOut**

Does not list the specified options.

**ALL**

Specifies the display of all the information associated with the CALC key.

**NONE**

Specifies the display of the name of the CALC key. NONE is meaningful only when the WITH clause is specified.
**DETAils**

Specifies the display of CALC key-specific descriptions.

**HISTORY**

Specifies the display of the date the CALC key was defined.

**AS COMMENTS**

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

**AS SYNTAX**

Outputs CALC key syntax which can be edited and resubmitted to the command facility.

**VERB CREATE/DISPLAY/DROP/PUNCH**

Specifies the verb with which the entity statement is to be displayed or punched. For example, if VERB CREATE is specified, the output of the DISPLAY/PUNCH statement is a CREATE statement; if VERB ALTER is specified, the output is an ALTER statement; and so on. The default is VERB CREATE.

---

**DISPLAY/PUNCH CONSTRAINT**

The DISPLAY/PUNCH CONSTRAINT statement displays or punches a referential constraint in the dictionary.

**Authorization**

To issue a DISPLAY/PUNCH CONSTRAINT statement, you must:

- Either hold the DISPLAY privilege on or own the referencing table in the constraint
- Hold the REFERENCES privilege on the referenced table in the constraint

**Syntax**

```sql
DISPLAY/PUNCH CONSTRAINT constraint-name [IN SCHEMA schema-name] [WITH ALL OUT WITHOUT DETAILS HISTORY] [AS COMMENTS SYNTAX] [VERB CREATE DISPLAY DROP PUNCH]
```
Parameters

**constraint-name**

Specifies the name of the referential constraint, within the current schema associated with your SQL session (if any), to display or punch.

**IN SCHEMA schema-name**

Specifies the name of the schema for the referential constraint if no schema is assigned with your SQL session.

**WITh**

Lists the requested information, in addition to the information that is always included, such as the entity occurrence name.

**WIThOut**

Does not list the specified options. *Other* options in effect through the **WITH** or **ALSO WITH** clauses in the current DISPLAY statement are displayed.
DISPLAY/PUNCH FUNCTION

ALSo WIth
Lists the requested information, in addition to the information requested in previously issued DISPLAY WITH and DISPLAY ALSO WITH statements for the named entity.

ALSo WIThOut
Does not list the specified options.

ALL
Specifies the display of all the information associated with the requested entity occurrence.

NONE
Specifies the display of the name of the requested entity occurrence. NONE is meaningful only when the WITH clause is specified.

DETails
Specifies the display of constraint-specific descriptions.

HIStory
Specifies the display of the date the constraint was created.

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

AS SYNTAX
Outputs constraint syntax which can be edited and resubmitted to the command facility.

VERB CREate/DISplay/DROp/PUNch
Specifies the verb with which the entity statement is to be displayed or punched. For example, if VERB CREATE is specified, the output of the DISPLAY/PUNCH statement is a CREATE statement; if VERB ALTER is specified, the output is an ALTER statement; and so on. The default is VERB CREATE.

DISPLAY/PUNCH FUNCTION

The DISPLAY/PUNCH FUNCTION statement lets you display or punch a function.

Authorization

To issue a DISPLAY/PUNCH FUNCTION statement, you must hold the DISPLAY privilege for the named function.
Syntax

```
DISPLAY    FUNction  function-name  FULL|PHYsical|
          PUNch        schema-name.

WITH
    WITHOUT    ALL
    Also WITH   NONE
    ALSO WITHOUT
```

Parameters

```
PARAMETERS

schema-name.
Identifies the SQL schema associated with the named function.
If you do not specify schema-name, then it defaults to the current schema
associated with your SQL session, if you enter the statement through the command
facility or execute it dynamically.

function-name
Specifies the name of the function to display or punch. function-name must be the
name of a function defined in the dictionary.

FULL
Directs CA IDMS to display all attributes of the function except physical attributes.

PHYsical
Directs CA IDMS to display all attributes of the function including its physical
attributes. This includes the function's synchronization timestamp.

WITH
Lists the requested information, in addition to the information that is always
included, such as the entity occurrence name.
**WITHOut**

Does not list the specified options. Other options in effect through the WITH or ALSO WITH clauses in the current DISPLAY statement display.

**ALSo WiTh**

Lists the requested information, in addition to the information requested in previously issued DISPLAY WITH and DISPLAY ALSO WITH statements for the named entity.

**ALSo WiTHOut**

Does not list the specified options.

**ALL**

Specifies the display of all the information associated with the requested entity occurrence.

**NONE**

Specifies the display of the name of the requested entity occurrence. NONE is meaningful only when you specify the WITH clause.

**DETails**

Specifies the display of entity-specific descriptions; for example, the length of a table.

**TIMestamp**

Specifies the display of the synchronization timestamp associated with the function.

**HIStory**

Specifies the display of the chronological account of an entity's existence, including PREPARED/REVISED BY specifications, date created, and date last updated.

**KEYs**

Specifies the display of all keys associated with the requested function.

**AS COMments**

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

**AS SYNtax**

Outputs function syntax which you can edit and resubmit to the command facility.

**VERB CREate/ALTer/DISplay/DROp/PUNch**

Specifies the verb with which the entity statement is to be displayed or punched. For example, if VERB CREATE is specified, the output of the DISPLAY/PUNCH statement is a CREATE statement; if VERB ALTER is specified, the output is an ALTER statement; and so on. The default is VERB CREATE.
Example

```sql
DISPLAY FUNCTION FIN.UDF_FUNBONUS FULL PHYSICAL;
```

**DISPLAY/PUNCH INDEX**

The DISPLAY/PUNCH INDEX statement displays or punches an index from the dictionary.

**Authorization**

To issue a DISPLAY/PUNCH INDEX statement, you must either own or have the DISPLAY privilege on the table on which the index is defined.

**Syntax**

```
DISPLAY INDEX index-name ON table-name
```

- `index-name` specifies the name of an index to display or punch. `index-name` must be the name of an index in the dictionary.
- `ON table-name` specifies the table on which the named index is defined.
- `INDEX` is the default flag.

**Parameters**

- `index-name` specifies the name of an index to display or punch. `index-name` must be the name of an index in the dictionary.
- `ON table-name` specifies the table on which the named index is defined.
- `schema-name` identifies the schema associated with the named table.

If you do not specify `schema-name`, it defaults to the current schema associated with your SQL session, if the statement is entered through the command facility or executed dynamically.
**WITH**

Lists the requested information, in addition to the information that is always included, such as the entity occurrence name.

**WITHOUT**

Does not list the specified options. Other options in effect through the WITH or ALSO WITH clauses in the current DISPLAY statement are displayed.

**ALSO WITH**

Lists the requested information, in addition to the information requested in previously issued DISPLAY WITH and DISPLAY ALSO WITH statements for the named entity.

**ALSO WITHOUT**

Does not list the specified options.

**ALL**

Specifies the display of all the information associated with the requested index.

**NONE**

Specifies the display of the name of the requested index. NONE is meaningful only when the WITH clause is specified.

**DETAILS**

Specifies the display of index-specific descriptions.

**HISTORY**

Specifies the display of the date the index was created.

**AS COMMENTS**

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

**AS SYNTAX**

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

**VERB CREATE/DISPLAY/DROP/PUNCH**

Specifies the verb with which the entity statement is to be displayed or punched. For example, if VERB CREATE is specified, the output of the DISPLAY/PUNCH statement is a CREATE statement; if VERB ALTER is specified, the output is an ALTER statement; and so on. The default is VERB CREATE.

**FULL**

Directs CA IDMS to display all attributes of the function except physical attributes.

**PHYSICAL**

Directs CA IDMS to display all attributes of the function including its physical attributes. This includes the function's synchronization timestamp.
DISPLAY/PUNCH KEY

The DISPLAY/PUNCH KEY statement displays a table procedure key definition stored in the dictionary.

Authorization

To issue a DISPLAY KEY statement, you must either own or hold the ALTER privilege on the table procedure on which the key being displayed or punched is defined.

Syntax

```
DISPLAY KEY key-name ON procedure-name
WITH ALL AS Comments
WITHOUT DETails
ALSO WITHOUT HISTORY
VERb CREATE DROP PUNCH
```

Parameters

**key-name**

Specifies the name of a key on a table procedure.

**schema-name.**

Identifies the schema associated with the table procedure.

If you do not specify a `schema-name` it defaults to the current schema associated with your SQL session, if the statement is entered through the Command Facility or executed dynamically.

**procedure-name**

Specifies the name of the procedure or table procedure on which the key is defined. The `procedure-name` must identify a procedure or table procedure defined in the dictionary.

**WITH**

Lists the requested information, in addition to the information that is always included, such as the entity occurrence name.

**WITHOUT**

Does not list the specified options. Other options in effect through the WITH or ALSO WITH clauses in the current DISPLAY statement are displayed.
The DISPLAY/PUNCH PROCEDURE statement displays or punches a procedure.

**Authorization**

To issue a DISPLAY PROCEDURE statement, you must have the DISPLAY privilege for the named procedure.
Syntax

```
DISPLAY   PROCEDURE   schema-name.   proc-name   FULL   PHYSical
WITH      ALSO WITH    ALL        AS COMMENTS
    WITHOUT      DETails    HISTORY
        ALSO WITHOUT    KEYs

VERb      CREATE
    DISPLAY
    DROP
    PUNch
```

Parameters

`schema-name`.

Identifies the SQL schema associated with the named procedure.

If you do not specify `schema-name`, then it defaults to the current schema associated with your SQL session, if you enter the statement through the command facility or execute it dynamically.

`procedure-name`

Specifies the name of the procedure to display or punch. `Procedure-name` must be the name of a procedure defined in the dictionary.

`FULL`

Directs CA IDMS to display all attributes of the procedure except physical attributes.

`PHYSical`

Directs CA IDMS to display all attributes of the procedure including its physical attributes. This includes the procedure’s synchronization timestamp.

`WITH`

Lists the requested information, in addition to the information that is always included, such as the entity occurrence name.

`WITHOUT`

Does not list the specified options. Other options in effect through the WITH or ALSO WITH clauses in the current DISPLAY statement display.

`ALSo WITh`

Lists the requested information, in addition to the information requested in previously issued DISPLAY WITH and DISPLAY ALSO WITH statements for the named entity.
ALSo WIThOut

Does not list the specified options.

ALL

Specifies the display of all the information associated with the requested entity occurrence.

NONE

Specifies the display of the name of the requested entity occurrence. NONE is meaningful only when you specify the WITH clause.

DETails

Specifies the display of entity-specific descriptions; for example, the length of a table.

HIStory

Specifies the display of the chronological account of an entity's existence, including PREPARED/REVISED BY specifications, date created, and date last updated.

TIMestamp

Specifies the display of the synchronization timestamp for the procedure.

KEYs

Specifies the display of all keys associated with the requested procedure.

AS COMments

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

AS SYNTAX

Outputs procedure syntax which you can edit and resubmit to the command facility.

VERB CRete/DISplay/DRop/PUNch

Specifies the verb with which the entity statement is to be displayed or punched. For example, if VERB CREATE is specified, the output of the DISPLAY/PUNCH statement is a CREATE statement; if VERB ALTER is specified, the output is an ALTER statement; and so on. The default is VERB CREATE.

DISPLAY/PUNCH SCHEMA

The DISPLAY/PUNCH SCHEMA statement displays or punches an SQL schema in the dictionary.
Authorization

To issue a DISPLAY/PUNCH SCHEMA statement, you must have the DISPLAY privilege on the requested SQL schema.

Syntax
Parameters

**SCHeema name**

Specifies the SQL schema to display or punch.

*Schema-name* must be the name of the an SQL schema in the dictionary.

**FULl or ALL**

Directs CA IDMS to display all attributes of the schema except physical attributes.

**PHYSical**

Directs CA IDMS to display all attributes of the schema including its physical attributes. This includes table IDs, index IDs, and synchronization timestamps for functions, procedures, tables, table procedures, and views.

**WITh**

Lists the requested information, in addition to the information that is always included, such as the entity occurrence name.

**WIThOut**

Does not list the specified options. *Other* options in effect through the WITH or ALSO WITH clauses in the current DISPLAY statement are displayed.

**ALSo WITh**

Lists the requested information, in addition to the information requested in previously issued DISPLAY WITH and DISPLAY ALSO WITH statements for the named entity.

**ALSo WIThOut**

Does not list the specified options.
ALL
    Specifies the display of all the information associated with the requested entity occurrence.

NONE
    Specifies the display of the name of the requested entity occurrence. NONE is meaningful only when the WITH clause is specified.

DETAILS
    Specifies the display of SQL schema-specific descriptions.

HISTORY
    Specifies the display of the chronological account of an entity's existence, including PREPARED/REVISED BY specifications, date created, and date last updated.

CALC keys
    Specifies the display of all CALC keys associated with the requested SQL schema.

CONSTRAINTS
    Specifies the display of all constraints associated with the requested SQL schema.

INDEXES
    Specifies the display of all indexes associated with the requested SQL schema.

KEYS
    Specifies the display of all table procedure keys associated with the requested SQL schema.
DISPLAY/PUNCH TABLE

The DISPLAY/PUNCH TABLE statement displays or punches the definition of a base table from the dictionary.

Authorization

To issue a DISPLAY/PUNCH TABLE statement, you must either own or have the DISPLAY privilege on the named table.
Syntax

```
DISPLAY TABLE schema-name table-name [FULL PHYSical]

WITH
    WITHOUT
    ALSO WITH
    ALSO WITHOUT

AS COMments SYNtax

VERb
    CREATE
    ALTER
    DISPLAY
    DROP
    PUNCH

LIKE RECORD

WITH
    WITHOUT
    ALSO WITH
    ALSO WITHOUT

AS COMments SYNtax
```

In IDD Record Format with COBOL Elements

```
DISPLAY TABLE schema-name table-name

LIKE RECORD

WITH
    WITHOUT
    ALSO WITH
    ALSO WITHOUT

null INDICators
record ELEMENTs
record SYNonyms
```

Parameters

**TABle table-name**

Specifies the name of the table to display or punch. *table-name* must be the name of a table defined in the dictionary.

**schema-name.**

Identifies the SQL schema associated with the named table.

If you do not specify *schema-name*, it defaults to the current schema associated with your SQL session, if the statement is entered through the command facility or executed dynamically.

**FULL**

Directs CA IDMS to display all attributes of the table except physical attributes.

**PHysical**

Directs CA IDMS to display all attributes of the table including physical attributes. This includes the table's synchronization timestamp and table ID.

**WITH**

Lists the requested information, in addition to the information that is always included, such as the entity occurrence name.

**WITHOut**

Does *not* list the specified options. *Other* options in effect through the WITH or ALSO WITH clauses in the current DISPLAY statement are displayed.

**ALSo WITh**

Lists the requested information, in addition to the information requested in previously issued DISPLAY WITH and DISPLAY ALSO WITH statements for the named entity.

**ALSo WIThOut**

Does *not* list the specified options.
ALL

Specifies the display of all the information associated with the requested table.

NONE

Specifies the display of the name of the requested table. NONE is meaningful only when the WITH clause is specified.

DETAILS

Specifies the display of table-specific descriptions; for example, the length of a table.

HISTORY

Specifies the display of the chronological account of a table's existence, including PREPARED/REVISED BY specifications, date created, and date last updated.

CALC keys

Specifies the display of a CALC key associated with the requested table occurrence.

COLUMNS

Specifies the display of all columns associated with the requested table occurrence.

CONSTRAINTS

Specifies the display of all constraints where the requested table occurrence has been named.

INDEXES

Specifies the display of all indexes associated with the requested table occurrence.

TIMESTAMP

Specifies the display of the synchronization timestamp for the table.
VIEWS

Specifies the display of all views where the requested table occurrence participates.

AS COMMENTS

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

AS SYNTAX

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

VERB CREATE/ALTER/DISPLAY/DROP/PUNCH

Specifies the verb with which the entity statement is to be displayed or punched. For example, if VERB CREATE is specified, the output of the DISPLAY/PUNCH statement is a CREATE statement; if VERB ALTER is specified, the output is an ALTER statement; and so on. The default is VERB CREATE.

With COBOL Elements Parameters

LIKE RECORD

Specifies that you want IDD RECORD syntax, with its COBOL elements, listed for the named table. For sample uses, see "Usage" later in this section.

NULL INDICATORS

Specifies the display of COBOL elements defining NULL indicators for nullable columns.

RECORD ELEMENTS

Specifies the display of elements for the record syntax for the named table.

RECORD SYNONYMS

Specifies the display of record synonyms for the record syntax for the named table.

Usage

Using the LIKE RECORD Parameter

You can use the LIKE RECORD parameter to produce IDD record syntax for a named table, and then add the record syntax to a dictionary.

With the IDD record syntax for a table in the dictionary, CA ADS dialogs can include a work record definition for the table. This same record definition can be included in a map definition.
DISPLAY/PUNCH TABLE PROCEDURE

The DISPLAY/PUNCH TABLE PROCEDURE statement displays or punches a table procedure.

Authorization

To issue a DISPLAY TABLE PROCEDURE statement, you must have the DISPLAY privilege for the named table procedure.

Syntax

```
DISPLAY TABLE PROCEDURE [FULL] [PHYSICAL] [schema-name] [table-proc-name]

WITH [ALL] [SYNTAX]
    | [AS Comments] [DETAILS] [HISTORY] [TIMESTAMP] [KEYS]
    | [NONE]

VERB [CREATE]
    | [DISPLAY] [DROP] [PUNCH]
```
DISPLAY/PUNCH TABLE PROCEDURE

Parameters

\textit{schema-name}.  
Identifies the SQL schema associated with the named table procedure.  
If you do not specify \textit{schema-name}, it defaults to the current schema associated with your SQL session, if the statement is entered through the command facility or executed dynamically.

\textit{table-procedure-name}  
Specifies the name of the table procedure to display or punch.  
\textit{Table-procedure-name} must be the name of a table procedure defined in the dictionary.

\textbf{FULL}  
Directs CA IDMS to display all attributes of the procedure except physical attributes.

\textbf{PHYSical}  
Directs CA IDMS to display all attributes of the procedure including its physical attributes. This includes the procedure's synchronization timestamp.

\textbf{WITH}  
Lists the requested information, in addition to the information that is always included, such as the entity occurrence name.

\textbf{WITHOUT}  
Does not list the specified options. \textit{Other} options in effect through the WITH or ALSO WITH clauses in the current DISPLAY statement are displayed.
DISPLAY/PUNCH TABLE PROCEDURE

ALSo WIWith
Lists the requested information, in addition to the information requested in previously issued DISPLAY WITH and DISPLAY ALSO WITH statements for the named entity.

ALSo WIThout
Does not list the specified options.

ALL
Specifies the display of all the information associated with the requested entity occurrence.

NONE
Specifies the display of the name of the requested entity occurrence. NONE is meaningful only when the WITH clause is specified.

DETAILs
Specifies the display of entity-specific descriptions; for example, the length of a table.

HISTORY
Specifies the display of the chronological account of an entity's existence, including PREPARED/REVISED BY specifications, date created, and date last updated.

TIMESTAMP
Specifies the display of the synchronization timestamp for the table procedure.

KEYs
Specifies the display of all keys associated with the requested table procedure.

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

AS SYNTAX
Outputs table procedure syntax which can be edited and resubmitted to the command facility.

VERB CREATE/DISPLAY/DROP/PUNCH
Specifies the verb with which the entity statement is to be displayed or punched. For example, if VERB CREATE is specified, the output of the DISPLAY/PUNCH statement is a CREATE statement; if VERB ALTER is specified, the output is an ALTER statement; and so on. The default is VERB CREATE.
DISPLAY/PUNCH VIEW

The DISPLAY/PUNCH VIEW statement displays or punches a view.

Authorization

To issue a DISPLAY VIEW statement, you must either own the SQL schema where the view is defined or hold the DISPLAY privilege on the named view.

Syntax

```
DISPLAY/PUNCH VIEW

WITH
  WITHOUT
  ALSO_WITH
  ALSO WITHOUT

SCHEMA_NAME.

VIEW.

FULL
  PHYSICAL

WITH
  WITHOUT
  ALL
  NONE
  DETAILS
  HISTORY
  TIMESTAMP

AS
  COMMENTS

SYNTAX

VERB
  CREATE
  DISPLAY
  DROP
  PUNCH

In IDD Record Format with COBOL Elements
```

```
DISPLAY/PUNCH VIEW

LIKE RECORD

WITH
  WITHOUT
  ALL
  NONE
  RECORD ELEMENTS
  RECORD SYNONYMS

AS
  COMMENTS

SYNTAX

VERB
  CREATE
  ALTER
  DISPLAY
  DROP
  PUNCH
```
Parameters

**VIEW** *view-name*

Specifies the name of the view to display or punch.

**schema-name.**

Identifies the SQL schema associated with the named view.

If you do not specify *schema-name*, it defaults to the current schema associated with your SQL session, if the statement is entered through the command facility or executed dynamically.

**FULL**

Directs CA IDMS to display all attributes of the view except physical attributes.

**PHYSICAL**

Directs CA IDMS to display all attributes of the view including its physical attributes. This includes the view’s synchronization timestamp.

**WITH**

Lists the requested information, in addition to the information that is always included, such as the entity occurrence name.

**WITHOUT**

Does not list the specified options. Other options in effect through the WITH or ALSO WITH clauses in the current DISPLAY statement are displayed.

**ALSO WITH**

Lists the requested information, in addition to the information requested in previously issued DISPLAY WITH and DISPLAY ALSO WITH statements for the named entity.

**ALSO WITHOUT**

Does not list the specified options.

**ALL**

Specifies the display of all the information associated with the requested entity occurrence.
NONE
   Specifies the display of the name of the requested entity occurrence. NONE is meaningful only when the WITH clause is specified.

DETails
   Specifies the display of entity-specific descriptions; for example, the length of a table.

HIStory
   Specifies the display of the chronological account of an entity’s existence, including PREPARED/REVISED BY specifications, date created, and date last updated.

TIMestamp
   Specifies the display of the synchronization timestamp for the view.

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

AS SYNTAX
   Outputs view syntax which can be edited and resubmitted to the command facility.

VERB CREATE/DISPLAY/DROP/PUNCH
   Specifies the verb with which the entity statement is to be displayed or punched. For example, if VERB CREATE is specified, the output of the DISPLAY/PUNCH statement is a CREATE statement; if VERB ALTER is specified, the output is an ALTER statement; and so on. The default is VERB CREATE.

With COBOL Elements Parameters

LIKE RECORD
   Specifies that you want IDD RECORD syntax, with its columns as COBOL elements, listed for the named view. For sample uses, see "Usage" later in this section.

null INDICATORS
   Specifies the display of COBOL elements defining NULL indicators for nullable columns.

record ELEMENTS
   Specifies the display of elements for the record syntax for the named view.

record SYNONYMS
   Specifies the display of record synonyms for the record syntax for the named view.
Usage

Using the LIKE RECORD Parameter

The LIKE RECORD parameter produces IDD record syntax for the named view.

You can use this syntax to define a record definition for a view in the dictionary. CA ADS dialogs can then include it as a work record definition for the view. This same record definition can be included in a map definition.
Appendix I: Sample COBOL Procedure

Sample Procedure Definition

The following example shows an SQL-invoked procedure definition.

```
create procedure demoempl.get_bonus
  ( emp_id          unsigned numeric(4)        with default,
    bonus           unsigned numeric(10)       with default,
    currency_bonus char (3)                    with default )
external name getbonus
protocol idms;
```
Sample Procedure Program

The following example shows a sample procedure program written in COBOL. This program requires the SQL employee demo database.

*COBOL PGM SOURCE FOR GETBONUS
*RETRIEVAL
*DMALIST
IDENTIFICATION DIVISION.
PROGRAM-ID. GETBONUS.
AUTHOR DEFJE01.
INSTALLATION SYSTEM71.
DATE-WRITTEN 06/25/99

*-------------------------------------------------------------
* GETBONUS will return the sum of all bonus amounts for a      *
* given employee.                                             *
* Parameters:                                                 *
* EMP_ID: input parameter must contain employee id            *
* BONUS output parameter returns sum of bonus                *
* CURRENCY-BONUS output parameter returns currency symbol    *
* or 'ERR' in case of an error condition                      *
* These parameters are assumed to have been defined          *
* 'WITH DEFAULT' in the procedure definition, so that null    *
* indicators do not need to be defined and processed          *
*-------------------------------------------------------------
ENVIRONMENT DIVISION.
*
CONFIGURATION SECTION.
*SOURCE-COMPUTER. IBM WITH DEBUGGING MODE.
*
DATA DIVISION.
*
WORKING-STORAGE SECTION.
01 ERROR-STATUS PIC X(4).
*
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
77 EMP-ID PIC 9(4).
77 BONUS PIC 9(10).
EXEC SQL END DECLARE SECTION END-EXEC.
77 CURRENCY-BONUS PIC X(3).
* Procedure parameters
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
* Other parameters do not need to be specified
EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION USING EMP-ID, BONUS, CURRENCY-BONUS.
0000-MAINLINE.

MOVE '$' TO CURRENCY-BONUS.

EXEC SQL
SELECT SUM(BONUS_AMOUNT) INTO :BONUS
FROM DEMOEMPL.BENEFITS
WHERE EMP_ID = :EMP-ID
END-EXEC

IF SQLSTATE NOT = '00000'
  MOVE 'ERR' TO CURRENCY-BONUS.

EXIT PROGRAM.
STOP RUN.
Sample of Procedure Invocation

The first four examples are all equivalent. The last example returns an error indication.

call demoempl.get_bonus(1234);
EMP_ID      BONUS   CURRENCY_BONUS
1234        6530   $
1 row processed

call demoempl.get_bonus(emp_id = 1234);
EMP_ID      BONUS   CURRENCY_BONUS
1234        6530   $
1 row processed

select * from demoempl.get_bonus where emp_id = 1234;
EMP_ID      BONUS   CURRENCY_BONUS
1234        6530   $
1 row processed

select * from demoempl.get_bonus(emp_id = 1234);
EMP_ID      BONUS   CURRENCY_BONUS
1234        6530   $
1 row processed

call demoempl.get_bonus(0);
EMP_ID      BONUS   CURRENCY_BONUS
0           0       ERR
1 row processed
Appendix J: CA IDMS Scalar Functions

Overview

This appendix contains an alphabetical listing of the SQL scalar functions that come with CA IDMS. These scalar functions are either built-in or defined in the SYSCA schema as a user-defined function. It is worth knowing if an SQL scalar function is built-in or user-defined because there is a limit on the number of user-defined functions that can be invoked in a single SQL statement.

Note: For more information, see Syntactic Limits.

The table notations are coded as follows:
- B = The function is implemented as a true built-in function.
- U = The function is implemented as a user-defined function in the SYSCA schema.

Functions

<table>
<thead>
<tr>
<th>Notation</th>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>ABS(number)</td>
<td>Absolute value of number</td>
</tr>
<tr>
<td>U</td>
<td>ACOS(float)</td>
<td>Arccosine, in radians, of float</td>
</tr>
<tr>
<td>U</td>
<td>ASIN(float)</td>
<td>Arcsine, in radians, of float</td>
</tr>
<tr>
<td>U</td>
<td>ATAN(float)</td>
<td>Arctangent, in radians, of float</td>
</tr>
<tr>
<td>U</td>
<td>ATAN2(float1, float2)</td>
<td>Arctangent, in radians, of float2/float1</td>
</tr>
<tr>
<td>B</td>
<td>CAST(number or null, AS datatype)</td>
<td>Converts value-expression to a specified data type.</td>
</tr>
<tr>
<td>B</td>
<td>CEIL(number)</td>
<td>Smallest integer greater than or equal to number data type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Same as CEILING.)</td>
</tr>
<tr>
<td>B</td>
<td>CEILING(number)</td>
<td>Smallest integer greater than or equal to number.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Same as CEIL.)</td>
</tr>
<tr>
<td>B</td>
<td>CHAR(code)</td>
<td>Character with ASCII code value code, where code is between 0 and 255 other data types.</td>
</tr>
<tr>
<td>Notation</td>
<td>Function</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td>B</td>
<td>CHAR_LENGTH(number)</td>
<td>Length of the value in value expression. (Same as CHARACTER_LENGTH.)</td>
</tr>
<tr>
<td>B</td>
<td>CHARACTER_LENGTH(number)</td>
<td>Length of the value in value expression. (Same as CHAR_LENGTH.)</td>
</tr>
<tr>
<td>B</td>
<td>COALESCE(datatype)</td>
<td>Substitutes a value for a null value. (Same as VALUE.)</td>
</tr>
<tr>
<td>B</td>
<td>CONCAT(string1, string2)</td>
<td>Character string formed by appending string2 to string1; if a string is null, the result is DBMS-dependent</td>
</tr>
<tr>
<td>B</td>
<td>CONVERT(value, SQLtype)</td>
<td>Value converted to SQLtype, where SQLtype can be any valid SQL data type.</td>
</tr>
<tr>
<td>U</td>
<td>COS(float)</td>
<td>Cosine of float radians</td>
</tr>
<tr>
<td>U</td>
<td>COSH(float)</td>
<td>Hyperbolic cosine of float radians</td>
</tr>
<tr>
<td>U</td>
<td>COT(float)</td>
<td>Cotangent of float radians</td>
</tr>
<tr>
<td>B</td>
<td>CURDATE()</td>
<td>The current date as a date value</td>
</tr>
<tr>
<td>B</td>
<td>CURTIME()</td>
<td>The current local time as a time value</td>
</tr>
<tr>
<td>B</td>
<td>DATABASE()</td>
<td>Current database</td>
</tr>
<tr>
<td>B</td>
<td>DATE(date)</td>
<td>Obtains the date from the value in value expression</td>
</tr>
<tr>
<td>B</td>
<td>DAY(date)</td>
<td>An integer from 1 to 41 representing the day of the month in date (Same as DAYOFMONTH.)</td>
</tr>
<tr>
<td>U</td>
<td>DAYNAME(date)</td>
<td>A character string representing the day component of date; the name for the day is specific to the data source</td>
</tr>
<tr>
<td>B</td>
<td>DAYOFMONTH(date)</td>
<td>An integer from 1 to 41 representing the day of the month in date (Same as DAY.)</td>
</tr>
<tr>
<td>U</td>
<td>DAYOFWEEK(date)</td>
<td>An integer from 1 to 7 representing the day of the week in date; 1 represents Sunday</td>
</tr>
<tr>
<td>U</td>
<td>DAYOFYEAR(date)</td>
<td>An integer from 1 to 366 representing the day of the year in date</td>
</tr>
<tr>
<td>B</td>
<td>DAYS(date)</td>
<td>An integer representation of the date in value expression</td>
</tr>
<tr>
<td>Notation</td>
<td>Function</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td>B</td>
<td>DECIMAL(<em>number, precision, scale</em>)</td>
<td>Decimal representation of the value in value expression.</td>
</tr>
<tr>
<td>U</td>
<td>DEGREES(<em>number</em>)</td>
<td>Degrees in <em>number</em> radians</td>
</tr>
<tr>
<td>B</td>
<td>DIGITS(<em>number</em>)</td>
<td>Character string representation of the value in value expression.</td>
</tr>
<tr>
<td>U</td>
<td>EXP(<em>float</em>)</td>
<td>Exponential function of <em>float</em></td>
</tr>
<tr>
<td>B</td>
<td>FLOAT(<em>float</em>)</td>
<td>Floating point representation of the value in value expression.</td>
</tr>
<tr>
<td>U</td>
<td>FLOOR(<em>number</em>)</td>
<td>Largest integer less than or equal to <em>number</em></td>
</tr>
<tr>
<td>B</td>
<td>HEX(<em>string</em>)</td>
<td>Hexadecimal representation of the value in value expression.</td>
</tr>
<tr>
<td>B</td>
<td>HOUR(<em>time</em>)</td>
<td>An integer from 0 to 23 representing the hour component of <em>time</em></td>
</tr>
<tr>
<td>B</td>
<td>IFNULL(<em>expression, value</em>)</td>
<td>Value if <em>expression</em> is null; <em>expression</em> if not null</td>
</tr>
<tr>
<td>U</td>
<td>INSERT(<em>string1, start, length, string2</em>)</td>
<td>A character string formed by deleting <em>length</em> characters from <em>string1</em> beginning at <em>start</em>, and inserting <em>string2</em> into <em>string1</em> at <em>start</em></td>
</tr>
<tr>
<td>B</td>
<td>INTEGER(<em>number</em>)</td>
<td>Integer representation of the value in value expression</td>
</tr>
<tr>
<td>B</td>
<td>LCASE(<em>string</em>)</td>
<td>Converts all uppercase characters in <em>string</em> to lowercase. (Same as LOWER.)</td>
</tr>
<tr>
<td>B</td>
<td>LEFT(<em>string, count</em>)</td>
<td>The <em>count</em> leftmost characters from <em>string</em></td>
</tr>
<tr>
<td>B</td>
<td>LENGTH(<em>string</em>)</td>
<td>Number of characters in <em>string</em>, excluding trailing blanks</td>
</tr>
<tr>
<td>B</td>
<td>LOCATE(<em>string1, string2[,start]</em>)</td>
<td>Position in <em>string2</em> of the first occurrence of <em>string1</em>, searching from the beginning of <em>string2</em>; if <em>start</em> is specified, the search begins from position <em>start</em> 0 is returned if <em>string2</em> does not contain <em>string1</em> Position 1 is the first character in <em>string2</em></td>
</tr>
<tr>
<td>U</td>
<td>LOG(<em>float</em>)</td>
<td>Base e logarithm of <em>float</em></td>
</tr>
<tr>
<td>U</td>
<td>LOG10(<em>float</em>)</td>
<td>Base 10 logarithm of <em>float</em></td>
</tr>
<tr>
<td>Notation</td>
<td>Function</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>B</td>
<td>LOWER(string)</td>
<td>Converts all uppercase characters in <code>string</code> to lowercase. (Same as LCASE.)</td>
</tr>
<tr>
<td>B</td>
<td>LTRIM(string)</td>
<td>Characters of <code>string</code> with leading blank spaces removed</td>
</tr>
<tr>
<td>B</td>
<td>MICROSECOND (timestamp)</td>
<td>Obtains the microsecond part of the value in value-expression.</td>
</tr>
<tr>
<td>B</td>
<td>MINUTE(time)</td>
<td>An integer from 0 to 59 representing the minute component of <code>time</code></td>
</tr>
<tr>
<td>U</td>
<td>MOD(integer1, integer2)</td>
<td>Remainder for <code>integer1/integer2</code></td>
</tr>
<tr>
<td>B</td>
<td>MONTH(date)</td>
<td>An integer from 1 to 12 representing the month component of <code>date</code></td>
</tr>
<tr>
<td>U</td>
<td>MONTHNAME(date)</td>
<td>A character string representing the month component of <code>date</code>; the name for the month is specific to the data source</td>
</tr>
<tr>
<td>B</td>
<td>NOW()</td>
<td>A timestamp value representing the current date and time</td>
</tr>
<tr>
<td>B</td>
<td>OCTET_LENGTH(number)</td>
<td>Obtains the length in bytes of the value in value-expression.</td>
</tr>
<tr>
<td>U</td>
<td>PI()</td>
<td>The constant pi</td>
</tr>
<tr>
<td>B</td>
<td>POSITION(string1, string2[,start])</td>
<td>Position in <code>string2</code> of the first occurrence of <code>string1</code>, searching from the beginning of <code>string2</code>; if <code>start</code> is specified, the search begins from position <code>start</code>; 0 is returned if <code>string2</code> does not contain <code>string1</code> Position 1 is the first character in <code>string2</code>.</td>
</tr>
<tr>
<td>U</td>
<td>POWER(number, power)</td>
<td><code>Number</code> raised to (integer) <code>power</code></td>
</tr>
<tr>
<td>B</td>
<td>PROFILE(string)</td>
<td>Obtains the value associated with an attribute of the current user session.</td>
</tr>
<tr>
<td>U</td>
<td>QUARTER(date)</td>
<td>An integer from 1 to 4 representing the quarter in <code>date</code>; 1 represents January 1 through March 31</td>
</tr>
<tr>
<td>U</td>
<td>RADIANS(number)</td>
<td>Radians in <code>number</code> degrees</td>
</tr>
<tr>
<td>U</td>
<td>RAND(integer)</td>
<td>Random floating point for seed <code>integer</code></td>
</tr>
<tr>
<td>U</td>
<td>REPEAT(string, count)</td>
<td>A character string formed by repeating <code>string</code> <code>count</code> times</td>
</tr>
<tr>
<td>Notation</td>
<td>Function</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td>U</td>
<td>REPLACE(string1, string2, string3)</td>
<td>Replaces all occurrences of string2 in string1 with string3</td>
</tr>
<tr>
<td>U</td>
<td>RIGHT(string, count)</td>
<td>The count rightmost characters in string</td>
</tr>
<tr>
<td>U</td>
<td>ROUND(number, places)</td>
<td>Number rounded to places places</td>
</tr>
<tr>
<td>B</td>
<td>RTRIM(string)</td>
<td>The characters of string with no trailing blanks</td>
</tr>
<tr>
<td>B</td>
<td>SECOND(time)</td>
<td>An integer from 0 to 59 representing the second component of time</td>
</tr>
<tr>
<td>U</td>
<td>SIGN(number)</td>
<td>-1 to indicate number is less than 0 0 to indicate number is equal to 0 1 to indicate number is greater than 0</td>
</tr>
<tr>
<td>U</td>
<td>SIN(float)</td>
<td>Sine of float radians</td>
</tr>
<tr>
<td>U</td>
<td>SINH(float)</td>
<td>Hyperbolic sine of float radians</td>
</tr>
<tr>
<td>U</td>
<td>SPACE(count)</td>
<td>A character string consisting of count spaces</td>
</tr>
<tr>
<td>U</td>
<td>SQRT(float)</td>
<td>Square root of float</td>
</tr>
<tr>
<td>B</td>
<td>SUBSTR(string, start, length)</td>
<td>A character string formed by extracting length characters from string beginning at start</td>
</tr>
<tr>
<td>B</td>
<td>SUBSTRING(string, FROM start, FOR length)</td>
<td>A character string formed by extracting length characters from string beginning at start</td>
</tr>
<tr>
<td>U</td>
<td>TAN(float)</td>
<td>Tangent of float radians</td>
</tr>
<tr>
<td>U</td>
<td>TANH(float)</td>
<td>Hyperbolic tangent of float radians</td>
</tr>
<tr>
<td>B</td>
<td>TIME(time)</td>
<td>Obtains time from the value in value expression</td>
</tr>
<tr>
<td>B</td>
<td>TIMESTAMP(string1, string2)</td>
<td>Obtains timestamp from a value or pair of values</td>
</tr>
<tr>
<td>B</td>
<td>TRIM(orientation, string)</td>
<td>Removes leading and/or trailing pad characters from CHARACTER or VARCHAR value expressions</td>
</tr>
<tr>
<td>U</td>
<td>TRUNCATE(number, places)</td>
<td>Number truncated to places places</td>
</tr>
<tr>
<td>B</td>
<td>UCASE(string)</td>
<td>Converts all lowercase characters in string to uppercase. (Same as UPPER.)</td>
</tr>
<tr>
<td>Notation</td>
<td>Function</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>B</td>
<td>UPPER(string)</td>
<td>Converts all lowercase characters in string to uppercase. (Same as UCASE.)</td>
</tr>
<tr>
<td>B</td>
<td>USER()</td>
<td>Current user</td>
</tr>
<tr>
<td>B</td>
<td>VALUE(datatype)</td>
<td>Substitutes a value for a null value. (Same as COALESCE.)</td>
</tr>
<tr>
<td>B</td>
<td>VARGRAPHIC(string)</td>
<td>Obtains graphic string representation of a character string</td>
</tr>
<tr>
<td>U</td>
<td>WEEK(date)</td>
<td>An integer from 1 to 53 representing the week of the year in date</td>
</tr>
<tr>
<td>B</td>
<td>YEAR(date)</td>
<td>An integer representing the year component of date</td>
</tr>
</tbody>
</table>
Appendix K: Sample COBOL Function

Sample Function Definition

The following example illustrates an SQL-invoked function definition:

```sql
CREATE FUNCTION FIN.UDF_FUNBONUS
  ( F_EMP_ID                           DECIMAL(4) )
  RETURNS DECIMAL(10)
  EXTERNAL NAME FUNBONUS PROTOCOL IDMS
  DEFAULT DATABASE CURRENT
  USER MODE
  LOCAL WORK AREA 0
;
```
Sample Function Program

The following example shows a sample SQL function program written in COBOL. This program requires the SQL employee demo database.

*COBOL PGM SOURCE FOR FUNBONUS
*RETRIEVAL
*DMLIST
IDENTIFICATION DIVISION.
PROGRAM-ID. FUNBONUS.
AUTHOR. DEFJE01.
INSTALLATION. SYSTEM71.
DATE-WRITTEN. mm/dd/yyyy.

*-------------------------------------------------------
* CA IDMS SQL                    nn.n                            *
*                                              *
* FUNBONUS implements the SQL function FUNBONUS      *
*                                              *
*-------------------------------------------------------
ENVIRONMENT DIVISION.
* CONFIGURATION SECTION.
*SOURCE-COMPUTER. IBM WITH DEBUGGING MODE.
* DATA DIVISION.
* WORKING-STORAGE SECTION.
*-------------------------------------------------------
EXEC SQL BEGIN DECLARE SECTION END-EXEC.

EXEC SQL
   INCLUDE TABLE FIN.UDF_FUNBONUS NO STRUCTURE
END-EXEC.

EXEC SQL END DECLARE SECTION END-EXEC.
The following example illustrates invoking the SQL function defined earlier:

```
SELECT EMP_ID, FIN.UDF_FUNBONUS(EMP_ID)
FROM DEMOEMPL.EMPLOYEE
WHERE EMP_ID = 3411
*+ EMP_ID USER_FUNC
*+ ------ ----------
*+ 3411 5100
*+
1 row processed
```
Appendix L: Sample CA ADS Procedure

SQL Procedure Example

The following SQL-invoked procedure, GET_PROC_AREA, writes any supplied message in a global area. The contents of the global area are shown when no input is supplied. The procedure definition is given next:

```
CREATE PROCEDURE DEFJE01.GET_PROC_AREA
  ( IN_AREA            CHARACTER (25),
    GLOBAL_AREA        CHARACTER (25)
  )
  EXTERNAL NAME GETPAREA
  PROTOCOL ADS
  SYSTEM MODE
  LOCAL WORK AREA 0
  GLOBAL WORK AREA 25 KEY GGLA
;
```

Work Records

To access the procedure parameters, the CA ADS dialog should include the <schema>.<procedure_name> as a work record. This record does not reside in the dictionary; it is automatically constructed by the CA ADS dialog compiler (ADSC or ADSOBCOM) when the dialog is compiled. The following DDDL syntax defines the global work area record:

```
ADD RECORD NAME GETPAREA-SQLPROC-GLOBAL-AREA.
  03 AREA-C                     PIC   X OCCURS 25.
```

The work records included in the mapless dialog GETPAREA are provided next:

- DEFJE01.GET_PROC_AREA
- GETPAREA-SQLPROC-GLOBAL-AREA
The premap process performs the actions of the SQL-invoked procedure. The premap process for the sample procedure is provided next:

ADD
PROCESS NAME IS GETAREA PROC VERSION IS 1
   PUBLIC ACCESS IS ALLOWED FOR ALL
   PROCESS SOURCE FOLLOWS
IF IN-AREA-I GE 0
   THEN
      D0.
      MOVE 0 TO GLOBAL-AREA-I.
      MOVE IN-AREA TO GLOBAL-AREA.
      MOVE IN-AREA TO GETAREA-SQLPROC-GLOBAL-AREA.
      MOVE 'WRITING TO GLOBAL AREA' TO IN-AREA.
   END.
ELSE
   D0.
   MOVE 0 TO IN-AREA-I.
   MOVE 'READING FROM GLOBAL-AREA' TO IN-AREA.
   MOVE 0 TO GLOBAL-AREA-I.
   MOVE GETAREA-SQLPROC-GLOBAL-AREA TO GLOBAL-AREA.
   END.
LEAVE ADS.
MSEND
Procedure Invocation

The GET_PROC_AREA invocation is given below. The first example illustrates writing to the global area:

```
CALL DEFJE01.GET_PROC_AREA ('HELLO FROM ADS DIALOG');
*+
*+ IN_AREA                      GLOBAL_AREA
*+  -       -                   ---------------
*+ WRITING TO GLOBAL AREA       HELLO FROM ADS DIALOG
*+
*+ 1 row processed
```

The second example illustrates reading from the global area:

```
CALL DEFJE01.GET_PROC_AREA ();
*+
*+ IN_AREA                      GLOBAL_AREA
*+  -       -                   ---------------
*+ READING FROM GLOBAL_AREA     HELLO FROM ADS DIALOG
*+
*+ 1 row processed
```
Appendix M: Sample CA ADS Function

SQL Function Example

The first example involves the SQL-invoked function ASIND, which returns the arcsine in degrees of the supplied value. The SQL function is implemented using a CA ADS dialog that invokes the CA ADS built-in function ARCSINE-DEGREES().

The SQL function is provided next:

```
CREATE FUNCTION DEFJE01.ASIND
  ( ARG               DOUBLE PRECISION)
RETURNS   DOUBLE PRECISION
EXTERNAL NAME ASIND
PROTOCOL ADS
SYSTEM MODE
LOCAL WORK AREA 0
GLOBAL WORK AREA 0
;
```

Work Records

To access the function parameters, the CA ADS dialog should include the <schema>.<function_name> as a work record. This record does not reside in the dictionary; it is automatically constructed by the CA ADS dialog compiler (ADSC or ADSOBCOM) when the dialog is compiled. ADSO-SQLPROC-COM-AREA is a system-supplied record. The work records included in the mapless dialog ASIND are shown in the following example:

- DEFJE01.ASIND
- ADSO-SQLPROC-COM-AREA
Premap Process

The premap process performs the actions of the SQL-invoked function. The following example shows the premap process for the sample function:

ADD MODULE NAME IS ASIND-PROC VERSION IS 1
LANGUAGE PROCESS
PROCESS SOURCE follows
IF ARG LE 1.0
    THEN
        DO.
            MOVE 0 TO USER-FUNC-I
            MOVE ARCSINE-DEGREES(ARG) TO USER-FUNC
        END.
    ELSE
        DO.
            MOVE '38099' TO SQLPROC-SQLSTATE.
            MOVE 'Arg must be <= 1.0' to SQLPROC-MSG-TEXT.
        END.
        LEAVE ADS.
MSEND

Function Invocation

The SELECT clause is used to invoke the function. The first example illustrates a correctly executing function:

SELECT DEFJE01.ASIND (1)
    FROM SYSTEM.TABLE WHERE NAME = 'ASIND'
*+ USER_FUNC
*+ 9.0000000000000000E+01
*+ 1 row processed

The second example illustrates a function invocation that results in an error message.

SELECT DEFJE01.ASIND (2)
    FROM SYSTEM.TABLE WHERE NAME = 'ASIND'
*+ Status = -4   SQLSTATE = 38000   Messages follow:
*+ DB001075 C-4M321: Table Procedure ASIND exception 38099 ARG MUST BE <= 1.0
Appendix N: SQL Cache Tables

Overview

This appendix describes the table procedures that are used for displaying and controlling the SQL cache. It also provides some examples of how the DBA can display and control the cache. The SQL cache is used in conjunction with the dynamic SQL statement caching feature. Dynamic SQL and dynamic SQL statement caching is explained in the CA IDMS SQL Programming Guide.

Tables for Viewing, Monitoring, and Controlling the Cache

SQL is the Application Programming Interface (API) used to view, monitor, and change the cache and the cache configuration. Therefore, cache administration, configuration, and dynamic SQL cache monitoring is available in any environment that supports CA IDMS SQL, such as IDMSBCF, OCF, CA IDMS Visual DBA, and the CA IDMS SQL programs, among others.

This section describes three table procedures and one view of the SYSCA tables defined for dynamic SQL cache management.

**DSCCACHEOPT**

The DSCCACHEOPT table manages the SQL cache options.

<table>
<thead>
<tr>
<th>Column</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACHEMAXCNT</td>
<td>INTEGER</td>
<td>The maximum number of entries that the cache can contain.</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>CHAR(4)</td>
<td>Default for caching: ON/OFF. This specifies if caching is enabled or disabled for any connect name that does not appear in the EXCEPTCON column.</td>
</tr>
<tr>
<td>EXCEPTCNT</td>
<td>INTEGER</td>
<td>Count of rows in the DSCCACHEOPT relation with non-NULL value for the EXCEPTON column, in other words, the number of connect names in the list of exceptions.</td>
</tr>
<tr>
<td>EXCEPTCON</td>
<td>CHAR(8)</td>
<td>Connect name that forms an exception to the default caching.</td>
</tr>
</tbody>
</table>
Tables for Viewing, Monitoring, and Controlling the Cache

Note the following:

- After startup of an IDMS Central Version, DSCCACHEOPT reflects the parameters of the sysgen SQL CACHE statement. In absence of an SQL CACHE statement there will be no rows in DSCCACHEOPT and SQL caching will be disabled, but can be activated by inserting a DSCCACHEOPT row. Updates to the DSCCACHEOPT table will have no impact on the sysgen of the CV.

- In local mode when no DSCCACHEOPT row exists, a DSCCACHEOPT row will be automatically inserted with values derived from the SYSIDMS parameter SQL_CACHE_ENTRIES.

- There can be 0 to \( n \) rows in this table. If there are 0 rows, this means that SQL statement caching is not active and not defined to the system. If there are rows, then the first row will have non-NULL values for CACHEMAXCNT, DEFAULT and EXCEPTCNT and a NULL value for EXCEPTCON. The first row contains the main SQL cache parameters. Other rows in the DSCCACHEOPT relation will have only non-NULL values for the EXCEPTCON column. These rows form the list of exception connect names.

- You can issue select, insert, update and delete commands against DSCCACHEOPT.

- Deleting the first row automatically deletes all other rows and removes all SQL cache structures from the system, effectively disabling caching until a new DSCCACHEOPT row is inserted. Deleting other rows removes exception connect names from the exception list.

- Inserting a row is always possible. When one or more rows already exists, an insert can only specify a value for EXCEPTCON, this is the way to add connect names to the list. When no rows exist, the first insert must specify values for CACHEMAXCNT and DEFAULT. Other values are not allowed. A successful insertion of the first row enables SQL caching.

- Updating of CACHEMAXCNT and DEFAULT columns automatically applies to the first row only, so that no WHERE clause is needed to filter the first row. When CACHEMAXCNT is decreased, the entries in the SQL cache with the highest AGE (see the section, "DSCCACHE") are removed. Increase CACHEMAXCNT to allow the size of the cache to be increased. You cannot update EXCEPTCON for the first row. You cannot update EXCEPTCNT as this is automatically calculated.

- The size of the cache is specified in terms of number of entries. Each entry represents a single cached statement. The cache is allocated from the storage pool within a central version and from operating system storage in local mode. You can determine the amount of storage being consumed by the cache by selecting from the DSCCACHECTRL table.
**DSCCACHECTRL**

The DSCCACHECTRL table controls SQL caching.

<table>
<thead>
<tr>
<th>Column</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQUEST</td>
<td>CHAR</td>
<td>Future use</td>
</tr>
<tr>
<td>STATUS</td>
<td>CHAR</td>
<td>Future use</td>
</tr>
<tr>
<td>CACHEMAXCNT</td>
<td>INTEGER</td>
<td>Maximum count of entries</td>
</tr>
<tr>
<td>CACHECURCNT</td>
<td>INTEGER</td>
<td>Current count of entries used</td>
</tr>
<tr>
<td>CURRENT</td>
<td>INTEGER</td>
<td>Current entry</td>
</tr>
<tr>
<td>OLDEST</td>
<td>INTEGER</td>
<td>Oldest entry</td>
</tr>
<tr>
<td>STORAGEUSEKB</td>
<td>INTEGER</td>
<td>Total storage used by the cache</td>
</tr>
</tbody>
</table>

Note the following:
- There can be 0 rows or 1 row in this table. If no rows are present, no SQL statements have been cached.
- Only select and delete statements against this table are possible.
- Deleting the 1 row in DSCCACHECTRL clears the SQL cache structures. It does not disable caching, which is controlled through the DSCCACHEOPT table.

**DSCCACHE**

The DSCCACHE table represents the SQL cache. Each row is a cache entry.

<table>
<thead>
<tr>
<th>Column</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY</td>
<td>INTEGER</td>
<td>Non-unique key</td>
</tr>
<tr>
<td>LOCK</td>
<td>BINARY(4)</td>
<td>Lock word for access to entry</td>
</tr>
<tr>
<td>DBNAME</td>
<td>CHAR (8)</td>
<td>DBName of SQL session</td>
</tr>
<tr>
<td>DEFAULTSHEMA</td>
<td>CHAR (18)</td>
<td>Default schema of session if statement contains at least one unqualified table reference</td>
</tr>
<tr>
<td>USECNT</td>
<td>INTEGER</td>
<td>Usage count</td>
</tr>
<tr>
<td>AGE</td>
<td>INTEGER</td>
<td>A value used to determine which entry to purge from a full cache when a new entry is inserted. The longer an entry remains in the cache without being used, the higher its age.</td>
</tr>
<tr>
<td>Column</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>COMPILECOST</td>
<td>INTEGER</td>
<td>Compilation cost</td>
</tr>
<tr>
<td>ACCPLANSCANSCOST</td>
<td>FLOAT</td>
<td>Cost of scan in access plan</td>
</tr>
<tr>
<td>ACCPLANCPUCOST</td>
<td>FLOAT</td>
<td>Cost of CPU in access plan</td>
</tr>
<tr>
<td>ACCPLANROWCOST</td>
<td>FLOAT</td>
<td>Count of rows in access plan</td>
</tr>
<tr>
<td>EXECCOST</td>
<td>INTEGER</td>
<td>Cost of last execution of statement</td>
</tr>
<tr>
<td>COMPILECNT</td>
<td>INTEGER</td>
<td>Count of (re)compilations</td>
</tr>
<tr>
<td>COMPILESTAMP</td>
<td>TIMESTAMP</td>
<td>Timestamp of compilation</td>
</tr>
<tr>
<td>STMTSIZEx</td>
<td>INTEGER</td>
<td>Size of statement</td>
</tr>
<tr>
<td>STATEMENT</td>
<td>VARCHAR(8192)</td>
<td>Statement</td>
</tr>
<tr>
<td>SQLDIBSIZE</td>
<td>INTEGER</td>
<td>Size of SQLDIB</td>
</tr>
<tr>
<td>SQLCMD</td>
<td>INTEGER</td>
<td>Type of SQL command</td>
</tr>
<tr>
<td>SQLITCL</td>
<td>INTEGER</td>
<td>Combined Itree/TELL table length</td>
</tr>
<tr>
<td>SQLARG</td>
<td>INTEGER</td>
<td>Bit flags for argument usage</td>
</tr>
<tr>
<td>SQLOPT</td>
<td>INTEGER</td>
<td>Session options flags</td>
</tr>
<tr>
<td>SQLTBL</td>
<td>INTEGER</td>
<td>Length of tuple buffer row</td>
</tr>
<tr>
<td>SQLPBL</td>
<td>INTEGER</td>
<td>Length of parameter buffer</td>
</tr>
<tr>
<td>SQLCID</td>
<td>INTEGER</td>
<td>Cursor identifier</td>
</tr>
<tr>
<td>SQLSID</td>
<td>INTEGER</td>
<td>Section identifier</td>
</tr>
<tr>
<td>SQLNM1</td>
<td>CHAR(32)</td>
<td>Literal value 1</td>
</tr>
<tr>
<td>SQLNM2</td>
<td>CHAR(32)</td>
<td>Literal value 2</td>
</tr>
<tr>
<td>SQLITL</td>
<td>INTEGER</td>
<td>Size of Itree</td>
</tr>
<tr>
<td>SQLITBADDR</td>
<td>BINARY(4)</td>
<td>Address of Itree</td>
</tr>
<tr>
<td>RTREESIZE</td>
<td>INTEGER</td>
<td>Size of Rtree</td>
</tr>
<tr>
<td>RTREEOFFSET</td>
<td>INTEGER</td>
<td>Offset of Rtree for relocation purposes</td>
</tr>
<tr>
<td>RTEEDOFAOFF</td>
<td>INTEGER</td>
<td>Offset of DOFA in Rtree</td>
</tr>
<tr>
<td>RTREEADDR</td>
<td>BINARY(4)</td>
<td>Address of Rtree</td>
</tr>
<tr>
<td>FIBSIZE</td>
<td>INTEGER</td>
<td>Size of FIB</td>
</tr>
<tr>
<td>FIBADDR</td>
<td>BINARY(4)</td>
<td>Address of FIB</td>
</tr>
<tr>
<td>FOPSIZE</td>
<td>INTEGER</td>
<td>Size of FOP</td>
</tr>
</tbody>
</table>
### Appendix N: SQL Cache Tables

<table>
<thead>
<tr>
<th>Column</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSTSIZE</td>
<td>INTEGER</td>
<td>Size of GST</td>
</tr>
<tr>
<td>FOPADDR</td>
<td>BINARY(4)</td>
<td>Address of FOP</td>
</tr>
<tr>
<td>LASTUSER</td>
<td>CHAR(8)</td>
<td>Reserved</td>
</tr>
<tr>
<td>GLOBALCURSORNAME</td>
<td>CHAR(18)</td>
<td>Reserved</td>
</tr>
<tr>
<td>FCRC</td>
<td>BINARY(4)</td>
<td>FCRC flags</td>
</tr>
<tr>
<td>SQLDAADDR</td>
<td>BINARY(4)</td>
<td>Address of cached input SQLDA</td>
</tr>
<tr>
<td>STORAGEUSED</td>
<td>INTEGER</td>
<td>Size in bytes of used storage</td>
</tr>
</tbody>
</table>

Note the following:

- One row of this table represents one cached statement.
- Rows cannot be inserted or updated.
- Because of the size of the STATEMENT column in DSCCACHE and also because many of these columns are for internal use only, it is advisable to use a view on this table procedure. The supplied DSCCACHEV view (shown next) is an example of such a view.

The following acronyms are used in the previous table.

- Itree: A data structure that contains the internal input representation of an SQL statement
- Rtree: A data structure that contains the internal runtime instruction of an SQL statement. The Rtree is used by the SQL runtime engine IDMSHLDB.
- FIB: A data structure that contains runtime metadata.
- FOB/FOP: FIB objects list data structure
- GST: Global Security Table
- FCRC: Fixed part of Compiled Relational Command data structure
- SQLDA: the SQL Descriptor Area (SQLDA) is a data structure used to describe variable data passed as part of a dynamic SQL statement.
Examples of Displaying and Controlling the Cache

**DSCCACHEV**

SYSCA.DSCCACHEV is created during installation. It defines a view on the SYSCA.DSCCACHE table procedure as follows:

```sql
create view SYSCA.DSCCACHEV as
    select KEY, DBNAME, DEFAULTSCHEMA, USECNT, AGE,
        COMPILECNT as "#C", compilestamp
        , ACCPLANSCANCOST, ACCPLANCPUCOST
        , ACCPLANROWCNT, FIBSIZE, FIBADDR
        , SUBSTR(STMT, 1, 72) as STMT1
    from SYSCA.DSCCACHE;
```

You have the option to define your own views.

**Allowable Operations on DSCCACHE Tables**

<table>
<thead>
<tr>
<th>Type</th>
<th>DSCCHAOPT</th>
<th>DSCCACHECTRL</th>
<th>DSCCACHE</th>
<th>DSCCACHEV</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>INSERT</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPDATE</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELETE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Examples of Displaying and Controlling the Cache**
CACHE Options

To display the cache options:

Select * from SYSCA.DSCCACHEOPT;

| +-----------------+------|--------|-----+-----------------+
| CACHEMAXCNT    | DEFAULT | EXCEPTCNT | EXCEPTCON |
| +-----------------+------|--------|-----+-----------------+
| 1000           | OFF  | 2     | <null>     |
| <null>         | <null> | <null> | SYSTEM      |
| <null>         | <null> | <null> | APPLDICT    |

To change the default for caching:

Update SYSCA.DSCCACHEOPT set DEFAULT = 'ON';

To add the connect name 'TSTDICT' to the exception list:

Insert into SYSCA.DSCCACHEOPT (EXCEPTCON) values ('TSTDICT');

To remove the connect name 'SYSTEM' from the exception list:

Delete from SYSCA.DSCCACHEOPT where EXCEPTCON = 'SYSTEM';

To remove all the connect names from the exception list:

Delete from SYSCA.DSCCACHEOPT where EXCEPTCON is not null;

To decrease the number of entries in the cache from 1000 to 5:

Update SYSCA.DSCCACHEOPT set CACHEMAXCNT = 5;

Only the last 5 used entries will be kept in the cache.

To increase the number of entries in the cache from 5 to 9999:

Update SYSCA.DSCCACHEOPT set CACHEMAXCNT = 9999;

The cache will be extended with 9994 new slots.

To clear the SQL cache and remove all the SQL cache structures from the system, effectively disallowing any SQL caching:

Delete from SYSCA.DSCCACHEOPT;

To rebuild the SQL cache environment or to build the SQL cache environment in a system that has no SQL CACHE statement in its SYSGEN:

Insert into SYSCA.DSCCACHEOPT (CACHEMAXCNT, DEFAULT) values (1000, 'ON');
Insert into SYSCA.DSCCACHEOPT (EXCEPTCON) values ('APPLDICT');
Insert into SYSCA.DSCCACHEOPT (EXCEPTCON) values ('SYSTEM');
CACHE Control Parameters

To display cache control parameters:

Select * from SYSCA.DSCCACHECTRL;

<table>
<thead>
<tr>
<th>REQUEST</th>
<th>STATUS</th>
<th>CACHEMAXCNT</th>
<th>CACHECURCNT</th>
<th>CURRENT</th>
<th>OLDEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>1000</td>
<td>7</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

To clear the cache, but allow caching to continue as defined by the option in DSCCACHEOPT:

Delete from SYSCA.DSCCACHECTRL;
CACHE Entries

To display key columns of all cache entries:

```
Select * from SYSCA.DSCCACHEV;
```

<table>
<thead>
<tr>
<th>KEY</th>
<th>DBNAME</th>
<th>DEFAULTSCHEMA</th>
<th>USECNT</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>SYSDICT</td>
<td>&lt;null&gt;</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>SYSDICT</td>
<td>&lt;null&gt;</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>SYSDICT</td>
<td>&lt;null&gt;</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>SYSDICT</td>
<td>&lt;null&gt;</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>29</td>
<td>SYSDICT</td>
<td>&lt;null&gt;</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

```
#C COMPILESTAMP     FIBSIZE  FIBADDR
1  2002-09-04-10.05.20.740186         736  12ACG208
1  2002-09-04-10.07.20.009275        2528  12ACD088
1  2002-09-04-10.06.19.785231        2580  12ACB888
1  2002-09-04-10.02.39.729463        2580  12ACB888
1  2002-09-04-10.03.00.735305         736  12ABFD88
```

```
STMT1
```

Select * from SYSCA.DSCCACHEV where AGE > 1;

To display cache entries for DBNAME SYSDICT:

```
Select * from SYSCA.DSCCACHEV where DBNAME = 'SYSDICT';
```

To display cache entries for statements that use schema EMPNSQL:

```
Select * from SYSCA.DSCCACHEV where STMT1 like '%EMPNSQL.%';
```

To remove cache entries that use schema EMPNSQL:

```
Delete from SYSCA.DSCCACHE where STATEMENT like '%empnsq1.%';
```
Secure the Display and Changes

To secure the display of and any changes to SQL caching, the DSCCACHE tables (table procedures and views) must be secured using the standard CA IDMS security mechanism.

**Note:** The SQL cache contains SQL source statements, which might include confidential information.
Appendix O: Enhancing the Presentation of Access Strategy Information

Overview

This appendix provides the definition and data of a table, an index, and a view to enable presenting the access strategy information in an easy-to-read and understandable format. The definitions and data are collected in one SQL script that is installed as member EXPLDDL in the CA IDMS source library.
The following SQL definitions can be very helpful when using the EXPLAIN command.

These definitions add meaning to the result of the EXPLAIN command by creating easy-to-understand values for the abbreviated codes that EXPLAIN produces. When you run an EXPLAIN statement against an SQL command or Access Module, the result is placed in an SQL table called ACCESS PLAN or another name that you choose. After you create the ACCESS CODE table and PLANVIEW view, you can easily query any EXPLAIN results and readily determine what kind of access strategies will be used for your SQL statements. This is especially useful for finding SQL queries that will cause area sweeps or other access strategies that might suggest adding new indices or other tuning options to the database.

Before running the script it is assumed that a "current" schema has been set as follows:

```
SET SESSION CURRENT SCHEMA <schema-name>
```

Also this current schema must have been assigned with an appropriate DEFAULT AREA to accommodate for the ACCESS_CODE and ACCESS_PLAN tables.

Set up access plan code table:

```sql
create table ACCESS_CODE
(COLUMN smallint not null, -- column index
 CODE_NUM smallint, -- numeric code
 CODE_CHAR char, -- character code
 TEXT char(16) not null) -- display text
no default index
);
create index ACCESS_CODE_IX
on ACCESS_CODE
(COLUMN, CODE_NUM, CODE_CHAR)
clustered
;```

```sql
```
These are the codes for the COMMAND column:
insert into ACCESS_CODE values (1,8,NULL,'DECLARE');
insert into ACCESS_CODE values (1,9,NULL,'DELETE');
insert into ACCESS_CODE values (1,17,NULL,'INSERT');
insert into ACCESS_CODE values (1,25,NULL,'SELECT');
insert into ACCESS_CODE values (1,29,NULL,'UPDATE');

These are the codes for the STYPE column:
insert into ACCESS_CODE values (2,0,NULL,' '); 
insert into ACCESS_CODE values (2,1,NULL,'Table Access');
insert into ACCESS_CODE values (2,2,NULL,'NL Join');
insert into ACCESS_CODE values (2,3,NULL,'SM Join');
insert into ACCESS_CODE values (2,4,NULL,'Sort');
insert into ACCESS_CODE values (2,5,NULL,'Merge Group');
insert into ACCESS_CODE values (2,6,NULL,'OR List');
insert into ACCESS_CODE values (2,7,NULL,'Dbk (Sorted)');
insert into ACCESS_CODE values (2,8,NULL,'Dbk (Unsorted)');

These are the codes for the ACMODE column:
insert into ACCESS_CODE values (3,NULL,' ',' '); 
insert into ACCESS_CODE values (3,NULL,'A','Area Sweep');
insert into ACCESS_CODE values (3,NULL,'C','Calc');
insert into ACCESS_CODE values (3,NULL,'I','Index');
insert into ACCESS_CODE values (3,NULL,'M','Set Member');
insert into ACCESS_CODE values (3,NULL,'N','Insert');
insert into ACCESS_CODE values (3,NULL,'O','Set Owner');
insert into ACCESS_CODE values (3,NULL,'P','Procedure');
insert into ACCESS_CODE values (3,NULL,'R','Rowid Indx');
insert into ACCESS_CODE values (3,NULL,'S','Index Seql');
insert into ACCESS_CODE values (3,NULL,'T','Temp Seql');

These are the codes for the SORTC and SORTN columns:
insert into ACCESS_CODE values (4,NULL,' ',' '); 
insert into ACCESS_CODE values (4,NULL,'D','Distinct');
insert into ACCESS_CODE values (4,NULL,'G','Group');
insert into ACCESS_CODE values (4,NULL,'M','Merge Join');
insert into ACCESS_CODE values (4,NULL,'O','Order By');
-- This statement forces the definition of the ACCESS_PLAN table,
-- so that the following view creation can reference it.
EXPLAIN STATEMENT 'SELECT * FROM SYSTEM.TABLE' STATEMENT NUMBER 9999;

-- This is a sample view definition. The ST instance of ACCESS_CODE
-- decodes the STYPE column of ACCESS_PLAN. The AM instance decodes the
-- ACMODE column. The S1 and S2 instances decode the SORTC and SORTN
-- columns, respectively. The COMMAND column isn't included in the
-- view, but could be decoded by introducing another ACCESS_CODE
-- INSTANCE in the FROM clause along with the matching join factors.
-- A descending sort was chosen to force the high level joins to the
-- top of the output - in sort of a top down tree format.
cREATE VIEW PLANVIEW(SNO, QB, PB, ST, "STEP TYPE", PST, TSCHEMA,
   "ACCESS MODE", ACNAME, LFS, OSORT,
   ISORT, SQC) AS
   SELECT CAST(SECTION AS DEC(4)),
          CAST(QBLOCK AS DEC(3)),
          CAST(PBLOCK AS DEC(3)),
          CAST(STEP AS DEC(3)),
          ST.TEXT,
          CAST(PSTEP AS DEC(3)),
          SUBSTR(TSCHEMA, 1, 8),
          SUBSTR(TABLE, 1, 10),
          SUBSTR(AM.TEXT, 1, 10),
          ACNAME,
          LFS,
          SUBSTR(S1.TEXT, 1, 10),
          SUBSTR(S2.TEXT, 1, 10),
          SUBQC
   FROM ACCESS_PLAN, ACCESS_CODE ST, ACCESS_CODE AM, ACCESS_CODE S1,
       ACCESS_CODE S2
   WHERE ST.COLUMN = 2 AND ST.CODE_NUM = STYPE
       AND AM.COLUMN = 3 AND AM.CODE_CHAR = ACMODE
       AND S1.COLUMN = 4 AND S1.CODE_CHAR = SORTC
       AND S2.COLUMN = 4 AND S2.CODE_CHAR = SORTN
   ORDER BY 1, 2, 4 DESC;

-- Look at the access plans
SELECT * FROM PLANVIEW;
<table>
<thead>
<tr>
<th>ABS</th>
<th>DAYOFWEEK</th>
<th>IN</th>
<th>OUTER</th>
<th>TANH</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACOS</td>
<td>DAYOFYEAR</td>
<td>INDEX</td>
<td>PARAMETERS</td>
<td>THEN</td>
</tr>
<tr>
<td>ADD</td>
<td>DAYS</td>
<td>INDEXES</td>
<td>PATH</td>
<td>TIME</td>
</tr>
<tr>
<td>ALL</td>
<td>DBNAME</td>
<td>INDICATOR</td>
<td>PI</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>ALLOCATE</td>
<td>DBTABLE</td>
<td>INNER</td>
<td>POSITION</td>
<td>TINYINT</td>
</tr>
<tr>
<td>ALTER</td>
<td>DEC</td>
<td>INOUT</td>
<td>POWER</td>
<td>TO</td>
</tr>
<tr>
<td>AND</td>
<td>DECIMAL</td>
<td>INSERT</td>
<td>PRECISION</td>
<td>TRANSACTION</td>
</tr>
<tr>
<td>ANY</td>
<td>DECLARE</td>
<td>INT</td>
<td>PREPARE</td>
<td>TRIM</td>
</tr>
<tr>
<td>AREA</td>
<td>DEFAULT</td>
<td>INTEGER</td>
<td>PRESERVE</td>
<td>TRUNCATE</td>
</tr>
<tr>
<td>AS</td>
<td>DEGREES</td>
<td>INTERNAL</td>
<td>PRIVILEGES</td>
<td>UCASE</td>
</tr>
<tr>
<td>ASIN</td>
<td>DELETE</td>
<td>INTO</td>
<td>PROCEDURE</td>
<td>UNDO</td>
</tr>
<tr>
<td>ATAN</td>
<td>DESCRIPTOR</td>
<td>IS</td>
<td>PROFILE</td>
<td>UNION</td>
</tr>
<tr>
<td>ATAN2</td>
<td>DIGITS</td>
<td>ITERATE</td>
<td>PROGRAM</td>
<td>UNIQUE</td>
</tr>
<tr>
<td>BEGIN</td>
<td>DISTINCT</td>
<td>JOIN</td>
<td>PROTOCOL</td>
<td>UNSIGNED</td>
</tr>
<tr>
<td>BETWEEN</td>
<td>DMCL</td>
<td>JOURNAL</td>
<td>QUARTER</td>
<td>UNTIL</td>
</tr>
<tr>
<td>BIGINT</td>
<td>DO</td>
<td>KEYS</td>
<td>QUEUE</td>
<td>UPDATE</td>
</tr>
<tr>
<td>BIN</td>
<td>DOUBLE</td>
<td>LANGUAGE</td>
<td>RADIANS</td>
<td>UPPER</td>
</tr>
<tr>
<td>BINARY</td>
<td>DROP</td>
<td>LCASE</td>
<td>RAND</td>
<td>USE</td>
</tr>
<tr>
<td>BUFFER</td>
<td>DSNAME</td>
<td>LEAVE</td>
<td>READ</td>
<td>USER</td>
</tr>
<tr>
<td>BY</td>
<td>DYNAMIC</td>
<td>LEFT</td>
<td>REFERENCES</td>
<td>USERID</td>
</tr>
<tr>
<td>CALL</td>
<td>ELSE</td>
<td>LENGTH</td>
<td>RELEASE</td>
<td>USING</td>
</tr>
<tr>
<td>CASCADE</td>
<td>ELSEIF</td>
<td>LIKE</td>
<td>RENAME</td>
<td>VALUE</td>
</tr>
<tr>
<td>CASE</td>
<td>ENCODING</td>
<td>LOAD</td>
<td>REPEAT</td>
<td>VALUES</td>
</tr>
<tr>
<td>CAST</td>
<td>END</td>
<td>LOCAL</td>
<td>REPLACE</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>CATALOG</td>
<td>ESCAPE</td>
<td>LOCATE</td>
<td>RESIGNAL</td>
<td>VARGRAPHIC</td>
</tr>
<tr>
<td>CEIL</td>
<td>ESTIMATED</td>
<td>LOG</td>
<td>RESULT</td>
<td>VIEW</td>
</tr>
<tr>
<td>CEILING</td>
<td>EXCEPTION</td>
<td>LOG10</td>
<td>RETURN</td>
<td>VOLUME</td>
</tr>
<tr>
<td>CHAR</td>
<td>EXEC</td>
<td>LOGINT</td>
<td>RETURNS</td>
<td>WEEK</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>EXECUTE</td>
<td>LOOP</td>
<td>REVOKE</td>
<td>WHEN</td>
</tr>
<tr>
<td>CHARACTERS</td>
<td>EXISTS</td>
<td>LOWER</td>
<td>RIGHT</td>
<td>WHERE</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------------</td>
<td>------------------------</td>
<td>------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>CHARACTER_LENGTH</td>
<td>EXIT</td>
<td>LTRIM</td>
<td>ROLLBACK</td>
<td>WHILE</td>
</tr>
<tr>
<td>CHAR_LENGTH</td>
<td>EXP</td>
<td>MAINTAIN</td>
<td>ROUND</td>
<td>WITH</td>
</tr>
<tr>
<td>CHECK</td>
<td>EXPLAIN</td>
<td>MAP</td>
<td>RTRIM</td>
<td>WITHIN</td>
</tr>
<tr>
<td>CLOSE</td>
<td>EXTERNAL</td>
<td>MICROSECOND</td>
<td>SCHEMA</td>
<td>WITHOUT</td>
</tr>
<tr>
<td>COALESCE</td>
<td>FETCH</td>
<td>MICROSECONDS</td>
<td>SECOND</td>
<td>XMLELEMENT</td>
</tr>
<tr>
<td>COLUMN</td>
<td>FILE</td>
<td>MINUTE</td>
<td>SECOND</td>
<td>XMLPOINTER</td>
</tr>
<tr>
<td>COMMIT</td>
<td>FLOAT</td>
<td>MINUTES</td>
<td>SECONDS</td>
<td>XMLSERIALIZE</td>
</tr>
<tr>
<td>CONCAT</td>
<td>FLOOR</td>
<td>MOD</td>
<td>SEGMENT</td>
<td>YEAR</td>
</tr>
<tr>
<td>CONDITION</td>
<td>FOR</td>
<td>MONTH</td>
<td>SELECT</td>
<td>YEARS</td>
</tr>
<tr>
<td>CONNECT</td>
<td>FROM</td>
<td>MONTHNAME</td>
<td>SET</td>
<td></td>
</tr>
<tr>
<td>CONSTRAINT</td>
<td>FUNCTION</td>
<td>MONTHS</td>
<td>SIGN</td>
<td></td>
</tr>
<tr>
<td>CONTENT</td>
<td>GET</td>
<td>NO</td>
<td>SIGNAL</td>
<td></td>
</tr>
<tr>
<td>CONVERT</td>
<td>GLOBAL</td>
<td>NOT</td>
<td>SIN</td>
<td></td>
</tr>
<tr>
<td>COS</td>
<td>GOTO</td>
<td>NOW</td>
<td>SINH</td>
<td></td>
</tr>
<tr>
<td>COSH</td>
<td>GRANT</td>
<td>NULL</td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td>COT</td>
<td>GRAPHIC</td>
<td>NUM</td>
<td>SPACE</td>
<td></td>
</tr>
<tr>
<td>CREATE</td>
<td>GROUP</td>
<td>NUMBER</td>
<td>SPECIFIC</td>
<td></td>
</tr>
<tr>
<td>CURDATE</td>
<td>HANDLER</td>
<td>OCTET_LENGTH</td>
<td>SORT</td>
<td></td>
</tr>
<tr>
<td>CURRENT</td>
<td>HAVING</td>
<td>OF</td>
<td>STANDARD</td>
<td></td>
</tr>
<tr>
<td>CURSOR</td>
<td>HEX</td>
<td>OFFSET</td>
<td>STORAGE</td>
<td></td>
</tr>
<tr>
<td>CURTIME</td>
<td>HOUR</td>
<td>ON</td>
<td>SUBSTR</td>
<td></td>
</tr>
<tr>
<td>DATABASE</td>
<td>HOURS</td>
<td>OPEN</td>
<td>SUBSTRING</td>
<td></td>
</tr>
<tr>
<td>DATE</td>
<td>IDMS</td>
<td>OPTIMIZE</td>
<td>SYSTEM</td>
<td></td>
</tr>
<tr>
<td>DAY</td>
<td>IF</td>
<td>OR</td>
<td>TABLE</td>
<td></td>
</tr>
<tr>
<td>DAYNAME</td>
<td>IFNULL</td>
<td>ORDER</td>
<td>TABLESPACE</td>
<td></td>
</tr>
<tr>
<td>DAYOFMONTH</td>
<td>IMMEDIATE</td>
<td>OUT</td>
<td>TAN</td>
<td></td>
</tr>
</tbody>
</table>
## Appendix Q: CA ADS, COBOL, PL/I Data Types

<table>
<thead>
<tr>
<th>CA ADS PICTURE and USAGE clause</th>
<th>CA IDMS data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC X(n) USAGE DISPLAY</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>01 name</td>
<td></td>
</tr>
<tr>
<td>49 name-LEN PIC S9(4) COMP</td>
<td></td>
</tr>
<tr>
<td>49 name-TEXT PIC X(n)</td>
<td></td>
</tr>
<tr>
<td>PIC S9(p-s)V9(s) USAGE COMP-3</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>PIC 9(p-s)V9(s) USAGE COMP-3</td>
<td>UNSIGNED DECIMAL(p,s)¹</td>
</tr>
<tr>
<td>USAGE COMP-2</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>USAGE COMP-1</td>
<td>REAL</td>
</tr>
<tr>
<td>USAGE COMP-1</td>
<td>FLOAT</td>
</tr>
<tr>
<td>PIC S9(n) USAGE COMP (where n&lt;5)</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>PIC S9(n) USAGE COMP (where n&gt;4 and n&lt;10)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>PIC S9(n) USAGE COMP (where n&gt;9)</td>
<td>LONGINT or BIGINT</td>
</tr>
<tr>
<td>PIC S9(p-s)V9(s) USAGE DISPLAY</td>
<td>NUMERIC(p,s)</td>
</tr>
<tr>
<td>PIC 9(p-s)V9(s) USAGE DISPLAY</td>
<td>UNSIGNED NUMERIC(p,s)¹</td>
</tr>
<tr>
<td>PIC X(n) USAGE DISPLAY</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>PIC G(n) USAGE DISPLAY-1</td>
<td>GRAPHIC(n)¹</td>
</tr>
<tr>
<td>01 name</td>
<td>VARGRAPHIC(n)¹</td>
</tr>
<tr>
<td>49 name-LEN PIC S9(4) COMP</td>
<td></td>
</tr>
<tr>
<td>49 name-TEXT PIC G(n) DISPLAY-1</td>
<td></td>
</tr>
<tr>
<td>PIC X(10) USAGE DISPLAY</td>
<td>DATE</td>
</tr>
<tr>
<td>PIC X(8) USAGE DISPLAY</td>
<td>TIME</td>
</tr>
<tr>
<td>PIC X(26) USAGE DISPLAY</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>PIC X(8) USAGE DISPLAY</td>
<td>TID²</td>
</tr>
</tbody>
</table>

¹ Indicates that the data type is signed or unsigned.
Note: This data type is a CA IDMS extension of the SQL standard.

<table>
<thead>
<tr>
<th>COBOL PICTURE and USAGE clause</th>
<th>CA IDMS data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC X(n) USAGE DISPLAY</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>01 name</td>
<td></td>
</tr>
<tr>
<td>49 name-LEN PIC S9(4) COMP</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>49 name-TEXT PIC X(n)</td>
<td></td>
</tr>
<tr>
<td>PIC S9(p-s)V9(s) USAGE COMP-3</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>PIC 9(p-s)V9(s) USAGE COMP-3</td>
<td>UNSIGNED DECIMAL(p,s)¹</td>
</tr>
<tr>
<td>USAGE COMP-2</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>USAGE COMP-1</td>
<td>REAL</td>
</tr>
<tr>
<td>USAGE COMP-1</td>
<td>FLOAT</td>
</tr>
<tr>
<td>PIC S9(n) USAGE COMP</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>(where n&lt;5)</td>
<td></td>
</tr>
<tr>
<td>PIC S9(n) USAGE COMP</td>
<td>INTEGER</td>
</tr>
<tr>
<td>(where n&gt;4 and n&lt;10)</td>
<td></td>
</tr>
<tr>
<td>PIC S9(n) USAGE COMP</td>
<td>LONGINT or BIGINT</td>
</tr>
<tr>
<td>(where n&gt;9)</td>
<td></td>
</tr>
<tr>
<td>PIC S9(p-s)V9(s) USAGE DISPLAY</td>
<td>NUMERIC(p,s)</td>
</tr>
<tr>
<td>PIC 9(p-s)V9(s) USAGE DISPLAY</td>
<td>UNSIGNED NUMERIC(p,s)¹</td>
</tr>
<tr>
<td>PIC X(n) USAGE SQLBIN</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>PIC G(n) USAGE DISPLAY-1</td>
<td>GRAPHIC(n)¹</td>
</tr>
<tr>
<td>01 name</td>
<td></td>
</tr>
<tr>
<td>49 name-LEN PIC S9(4) COMP</td>
<td>VARGRAPHIC(n)¹</td>
</tr>
<tr>
<td>49 name-TEXT PIC G(n) DISPLAY-1</td>
<td></td>
</tr>
<tr>
<td>PIC X(10) USAGE DISPLAY</td>
<td>DATE</td>
</tr>
<tr>
<td>PIC X(8) USAGE DISPLAY</td>
<td>TIME</td>
</tr>
<tr>
<td>PIC X(26) USAGE DISPLAY</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>PIC X(8) USAGE SQLBIN</td>
<td>TID¹</td>
</tr>
</tbody>
</table>

Note: This data type is a CA IDMS extension of the SQL standard.
<table>
<thead>
<tr>
<th>Equivalent PL/I data type</th>
<th>CA IDMS data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (n)</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>CHAR (n) VAR</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>FIXED DECIMAL (p,s)</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>FLOAT BINARY (n)</td>
<td></td>
</tr>
<tr>
<td>where n &lt;= 24</td>
<td>REAL</td>
</tr>
<tr>
<td>where n &gt; 24</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>FLOAT DECIMAL (n)</td>
<td></td>
</tr>
<tr>
<td>where n &lt;= 6</td>
<td>REAL</td>
</tr>
<tr>
<td>where n &gt; 6</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>FIXED BINARY (15)</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>FIXED BINARY (31)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>CHAR (n)</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>GRAPHIC (n)</td>
<td>GRAPHIC(n)(^1)</td>
</tr>
<tr>
<td>GRAPHIC (n) VAR</td>
<td>VARGRAPHIC(n)(^1)</td>
</tr>
<tr>
<td>CHAR (10)</td>
<td>DATE</td>
</tr>
<tr>
<td>CHAR (8)</td>
<td>TIME</td>
</tr>
<tr>
<td>CHAR (26)</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>SQLBIN (n)</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>CHAR(8)</td>
<td>TID(^2)</td>
</tr>
</tbody>
</table>

**Note:** This data type is a CA IDMS extension of the SQL standard.
Appendix R: Third-Party Acknowledgment

Portions of this product include software developed by the Daniel Veillard. The libxml2 software is distributed in accordance with the following license agreement:

Copyright © 1998-2012 Daniel Veillard. All Rights Reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE DANIEL VEILLARD BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name of Daniel Veillard shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization from him.
Index

A
About Identifiers • 24
Accessing a Table Procedure • 618
Accessing Non-SQL-Defined Databases • 599
Aggregate-function • 117
aliases • 107
    in labeled-duration • 107
ALLOCATE CURSOR • 258
ALTER ACCESS MODULE • 261
ALTER CATALOG • 269
ALTER CONSTRAINT • 270
ALTER CONSTRAINT Parameters • 271
ALTER FUNCTION • 272
ALTER FUNCTION statement • 272
syntax • 272
ALTER INDEX • 277
ALTER PROCEDURE • 279
ALTER SCHEMA • 285
ALTER TABLE • 290
ALTER TABLE PROCEDURE • 300

B
BEGIN DECLARE SECTION • 305

C
CA ADS, COBOL, PL/I Data Types • 849
CA IDMS Scalar Functions • 124, 817
CA Technologies Product References • 3
CALL • 306, 561
CASE • 562
CLOSE • 313
COBOL/CA ADS SQLCA • 683
Coding Considerations • 15
COMMIT • 315
Comparison, Assignment, Arithmetic, and Concatenation Operations • 66
Compound Statement • 566
CONNECT • 317
Considerations for SQL-invoked External Routines • 647
Contact CA Technologies • 3
Contents of EXPLDDL • 844
Control Statements • 559
Correspondence between SQL and Non-SQL-defined Entities • 599
CREATE ACCESS MODULE • 320
CREATE CALC • 333
CREATE CONSTRAINT • 334
CREATE FUNCTION • 341
CREATE INDEX • 352
CREATE KEY • 357
CREATE PROCEDURE • 361
CREATE SCHEMA • 373
CREATE TABLE • 378
CREATE TABLE PROCEDURE • 387
CREATE TEMPORARY TABLE • 392
CREATE VIEW • 394

D
Data Type Limits • 671
Data Types • 53
Data Types and Null Values • 53
Database Name Inheritance • 653
Date/time Arithmetic • 110
DEALLOCATE PREPARE • 401
Debugging Procedures • 653
DECLARE CURSOR • 402
DECLARE EXTERNAL CURSOR • 406
Defining a Function • 640
Defining a Procedure • 630
Defining a Table Procedure • 618
Defining and Using Functions • 639
Defining and Using Procedures • 629
Defining and Using Table Procedures • 617
Definitions • 15
DELETE • 408
DESCRIBE • 413
DESCRIBE CURSOR • 416
DISPLACEMENT • 749
DISPLAY and PUNCH Operations • 770
DISPLAY and PUNCH Syntax • 769
DISPLAY/PUNCH ACCESS MODULE • 781
DISPLAY/PUNCH ALL Statement • 770
DISPLAY/PUNCH CALC KEY • 784
DISPLAY/PUNCH CONSTRAINT • 786
DISPLAY/PUNCH FUNCTION • 788
DISPLAY/PUNCH INDEX • 791
DISPLAY/PUNCH KEY • 793
Invoking a Function • 641
Invoking a Procedure • 631
ITERATE • 580

L
LEAVE • 583
Literals • 75
Local Variables • 81
Logical Data Limits • 671
LOOP • 585

M
Mapping Plain Text SQL to XML • 658
Mapping SQL Data Type Values to XML Schema Data Type Values • 660
Mapping SQL Identifier to XML • 658

N
nonsql-database name • 373
   specified in CREATE SCHEMA • 373
Null Values • 73

O
OPEN • 494
Overview • 207, 559, 691, 817, 833, 843

P
PL/I SQLCA • 684
Predicates and Search Condition • 207
Premap Process • 828, 832
PREPARE • 497
Procedure Invocation • 829
Procedure Parameters • 631

Q
Query Expressions • 240
Query Specifications • 231
Query Specifications, Subqueries, Query Expressions, and Cursor Specifications • 231

R
RELEASE • 501
REPEAT • 587
Representation of Date/Time Values • 63
RESIGNAL • 589
RESUME SESSION • 502
RETURN • 591
REVOKE All Table Privileges • 503
REVOKE Execution Privilege • 510
REVOKE SQL Definition Privileges • 506
REVOKE Table Access Privileges • 512
ROLLBACK • 516
Routine Parameters • 84
ROWID Pseudo-column • 95

S
Sample CA ADS Function • 831
Sample CA ADS Procedure • 827
Sample COBOL Function • 823
Sample COBOL Procedure • 813
Sample COBOL Table Procedure • 751
Sample Function Definition • 823
Sample Function Program • 824
Sample of Procedure Invocation • 816
Sample Procedure Definition • 813
Sample Procedure Program • 814
Sample Table Procedure Definition • 751
Sample Table Procedure Program • 752
Scalar Function • 121
scalar functions • 124
   EXP • 124
   TRUNCATE • 124
Secure the Display and Changes • 842
SELECT • 518
SET ACCESS MODULE • 534
SET Assignment • 592
SET host-variable Assignment • 535
SET SESSION • 536
SET TRANSACTION • 542
SIGNAL • 594
Special Considerations for SQL-invoked External Routines • 647
Special Registers • 93
SQL Access to Non-SQL Databases • 609
SQL Cache Tables • 833
SQL Comments • 18
SQL Communication Area • 675
SQL Control Statements • 560
SQL Descriptor Area • 685
SQL DML Statements Operating on Non-SQL-defined Records • 606
SQL Function Example • 831
SQL Procedure Example • 827
SQL Reserved Words • 847
SQL Schema Considerations • 604

Index 857
| SQL Standard Basis | 667 |
| SQLALN and SQLNALN | 689 |
| SQLCA | 675 |
| SQLCODE | 681 |
| SQLCODE and SQLCNRP Values | 681 |
| SQLCODE Error Values | 681 |
| SQLDA | 685 |
| SQLLEN | 686 |
| SQLNULL | 689 |
| SQLPRECISION | 688 |
| SQLSCALE | 688 |
| SQLSTATE | 676 |
| SQLSTATE Values | 676 |
| SQLTYPE | 687 |
| Statement Categories | 251 |
| Statements | 249 |
| Subqueries | 238 |
| Summary Comparison to SQL Standard | 667 |
| Summary of Limits | 671 |
| SUSPEND SESSION | 544 |
| Syntactic Limits | 673 |
| Syntax | 271 |
| ALTER CONSTRAINT | 271 |
| Syntax Diagram Conventions | 19 |
| SYSCA Objects | 743 |
| SYSTEM Tables and SYSCA Views | 691 |
| SYSTEM.AM | 691 |
| SYSTEM.AMDEP | 692 |
| SYSTEM.AREA | 692 |
| SYSTEM.BUFFER | 694 |
| SYSTEM.COLUMN | 695 |
| SYSTEM.CONSTKEY | 697 |
| SYSTEM.CONSTRAINT | 698 |
| SYSTEM.DBNAME | 704 |
| SYSTEM.DBSEGMENT | 706 |
| SYSTEM.DBSSC | 706 |
| SYSTEM.DBTABLE | 707 |
| SYSTEM.DMCL | 708 |
| SYSTEM.DMCLAREA | 709 |
| SYSTEM.DMCLFILE | 711 |
| SYSTEM.DMCALSEGMENT | 713 |
| SYSTEM.FILE | 714 |
| SYSTEM.FILEMAP | 716 |
| SYSTEM.INDEX | 717 |
| SYSTEM.INDEXKEY | 725 |
| SYSTEM.JOURNAL | 726 |
| SYSTEM.LOADHDR | 728 |
| SYSTEM.ORDERKEY | 729 |
| SYSTEM.SCHEMA | 730 |
| SYSTEM.SECTION | 732 |
| SYSTEM(SEGMENT | 733 |
| SYSTEM(SYMBOL | 734 |
| SYSTEM(SYNTAX | 736 |
| SYSTEM(TABLE | 737 |
| SYSTEM.VIEWDEF | 742 |
| T |
| Table Procedure Parameters | 619 |
| Tables for Viewing, Monitoring, and Controlling the Cache | 833 |
| Third-Party Acknowledgment | 853 |
| Transaction Sharing | 654 |
| TRANSFER OWNERSHIP | 545 |
| U |
| UPDATE | 546 |
| Using SQL Statements | 16 |
| V |
| Values and Value Expressions | 75 |
| W |
| When to Use a Procedure | 629 |
| When to Use a Table Procedure | 617 |
| When to Use a User-Defined Function | 639 |
| WHenever | 556 |
| WHILE | 597 |
| Work Records | 827, 831 |
| Writing a Table Procedure | 622 |
| Writing an External Function in COBOL, PL/I, or Assembler | 641 |
| Writing an External Procedure in COBOL, PL/I or Assembler | 634 |
| Writing External Functions as CA ADS Mapless Dialogs | 644 |
| Writing External Procedures as CA ADS Mapless Dialogs | 636 |
| X |
| XML Publishing | 655 |
| XML Publishing Using SQL | 655 |
| XML Value Functions | 181 |
| XMLSLICE Table Procedure | 663 |