Progress®



OpenEdge
SQL Reference

OpenEdge®

Table of Contents

Preface....19

Examples of SQL syntax diagrams....23

Long SQL syntax descriptions split across lines....24

Part I: SQL Reference....25

OpenEdge SQL statements....27

ALTER DATABASE SET PRO_ENABLE_LARGE_KEYS....29

ALTER DATABASE SET PRO_ENABLE_64BIT_SEQUENCES....30

ALTER GROUP....30

Authorization....31

Related Statements....31

ALTER SEQUENCE....31

Authorization....33

Related Statement....34

ALTER TABLE....34

Defining LOB columns....42

ALTER TABLE RENAME COLUMN....42

ALTER TABLE RENAME INDEX....43

ALTER TABLE DROP COLUMN....43

Converting a table to a multi-tenant table....44

Repairing a CRC mismatch error....45

Migrating data from existing unpartitioned tables to partitioned tables....46

ALTER TABLE syntax to add or drop partitions....62

Authorization....67

Related statements....67

Altering partitioned tables without partition schema definition....67

ALTER TENANT....69

Authorization....70

Related Statement....71

ALTER USER....71

Authorization....72

Related statements....72

AUDIT INSERT....72

AUDIT SET....73

Authorization....75

CALL....75

Authorization....75

Related statements....75
Column constraints....76
COMMIT....78
Related Statement....79
CONNECT AS CATALOG....79
Authorization....80
SQL Compliance....80
Related statements....80
CREATE DOMAIN....80
Authorization....81
Related Statement....81
CREATE GROUP....81
Authorization....83
Related Statement....83
CREATE INDEX....83
Creating indexes for partitioned tables....86
Authorization....88
Related statements....88
CREATE PROCEDURE....88
Authorization....90
Related statements....90
CREATE SEQUENCE....90
Authorization....92
Related Statements....93
CREATE SUPER-TENANT....93
Authorization....93
Related Statement....93
CREATE SYNONYM....94
Authorization....94
Related Statement....94
CREATE TABLE....94
Authorization....107
Related Statement....107
CREATE TENANT....107
Authorization....110
Related Statement....110
CREATE TRIGGER....110
Authorization....113
Related statement....113
CREATE USER....113
Authorization....115
Related statements....115
CREATE VIEW....115
Authorization....117
Related statements....117

DELETE 117
Authorization....118
Related statements....118
DISCONNECT CATALOG....118
Authorization....119
SQL Compliance....119
Related statements....119
DROP DOMAIN....119
Authorization....120
Related Statement....120
DROP GROUP....120
Authorization....121
Related statements....121
DROP INDEX 121
Authorization....122
Related statement....122
DROP PROCEDURE 122
Authorization....123
Related statement....123
DROP SEQUENCE 123
Authorization....123
Related Statement....123
DROP SYNONYM 124
Authorization....124
Related statement....124
DROP TABLE 124
Authorization....126
Related statement....126
DROP TENANT 126
Authorization....127
Related Statement....127
DROP TRIGGER 127
Authorization....128
Related statement....128
DROP USER 128
Authorization....129
Related statements....129
DROP VIEW 129
Authorization....130
Related statement....130
GRANT 130
Authorization....135
Related statement....135
INSERT 135
Authorization....139

Related statement....139
LOCK TABLE....139
Authorization....140
Related statements....141
REVOKE....141
Syntax to revoke privileges on specific tables and views....143
Authorization....145
Related statement....145
ROLLBACK....145
Authorization....146
Related statements....146
SELECT....146
COLUMN_LIST clause....148
TOP clause....151
FROM clause....152
WHERE clause....154
GROUP BY clause....154
HAVING clause....156
ORDER BY clause....157
OFFSET and FETCH clauses....158
WITH clause....160
FOR UPDATE clause....161
Authorization....162
Related Statements....162
SET CATALOG....162
Authorization....163
SQL Compliance....163
Related statements....163
SET DATABASE DEFAULT AREA....163
SET PRO_CONNECT LOG....164
SET PRO_CONNECT QUERY_TIMEOUT....165
SET PRO_SERVER QUERY_TIMEOUT....166
SET PRO_SERVER LOG....167
SET ROWCOUNT....169
Related Statement....171
SET SCHEMA....171
Authorization....172
SET TRANSACTION ISOLATION LEVEL....172
Authorization....173
Related statements....173
SHOW CATALOGS....173
Authorization....174
SQL Compliance....174
Related statements....174
SHOW DATABASE DEFAULT AREA....174

SHOW GROUP....174
Authorization....176
Related Statement....176
SHOW PARTITION....176
Authorization....178
SHOW ROWCOUNT....178
Related Statement....178
SHOW TENANT....178
Authorization....179
Related Statement....179
SHOW ENCRYPT ON....180
Authorization....180
SQL Compliance....180
Related statements....180
Table constraints....181
UPDATE....183
Authorization....186
Related statements....186
UPDATE STATISTICS....186
Authorization....190

OpenEdge SQL functions....191

About OpenEdge SQL functions....194
Aggregate functions....195
Scalar functions....195
ABS....195
ACOS....196
ADD_MONTHS....196
ASCII....197
ASIN....198
ATAN....198
ATAN2....199
AVG....200
CASE....200
CAST....203
CDC_get_changed_columns....204
CDC_is_column_changed....205
CEILING....205
CHAR....206
CHR....207
COALESCE....207
CONCAT....208
CONVERT (ODBC compatible)....209
CONVERT (Progress extension)....209

COS 210
COUNT 211
CURDATE 211
CURTIME 212
CURRVAL 212
DATABASE 213
DAYNAME 214
DAYOFMONTH 215
DAYOFWEEK 215
DAYOFYEAR 216
DB_NAME 216
DECODE 217
DEGREES 218
EXP 218
FLOOR 219
GREATEST 219
HOUR 220
IFNULL 221
INITCAP 221
INSERT....222
INSTR 223
ISOWEEKDAY....224
ISOWEEK....224
ISOYEAR....225
LAST_DAY 226
LCASE 226
LEAST 227
LEFT 227
LENGTH 228
LOCATE 229
LOG10 229
LOWER 230
LPAD 231
LTRIM 231
MAX 232
MIN 233
MINUTE 233
MOD 234
MONTH 234
MONTHNAME 235
MONTHS_BETWEEN 235
NEXT_DAY 236
NEXTVAL....237
NOW 238
NULLIF 238

NVL 239
PI 239
POWER 240
PREFIX 240
PRO_ARR_DESCAPE function....241
PRO_ARR_ESCAPE function....242
PRO_ELEMENT function....243
QUARTER 245
RADIANS 246
RAND 246
REPEAT 247
REPLACE 248
RIGHT 248
ROUND 249
ROWID 250
RPAD 251
RTRIM 252
SECOND 253
SIGN 253
SIN 254
SQRT 254
SUBSTR 255
SUBSTRING (ODBC compatible)....256
SUFFIX 257
SUM 258
SYSDATE 259
SYSTIME 259
SYSTIMESTAMP 260
TAN 260
TO_CHAR 261
TO_DATE 262
TO_NUMBER 262
TO_TIME 263
TO_TIMESTAMP 263
TRANSLATE 264
UCASE 265
UPPER 266
USER 266
WEEK 267
YEAR 267

OpenEdge SQL reserved words....269

OpenEdge SQL error messages.....275

Overview.....275
Error codes, SQLSTATE values, and messages.....276

OpenEdge SQL system limits.....293

OpenEdge SQL system catalog tables.....295

Overview of system catalog tables.....296
SYSTABLES 299
SYSCOLUMNS 299
SYSINDEXES 300
SYSCALCTABLE 301
SYSNCHARSTAT 301
SYSCOLAUTH 301
SYSCOLSTAT 302
SYSCOLUMNS_FULL 303
SYSDATATYPES 304
SYSDATESTAT 305
SYSDBAUTH 305
SYSFLOATSTAT 306
SYSIDXSTAT 306
SYSINTSTAT 306
SYSNUMSTAT 307
SYSPROCBIN 307
SYSPROCCOLUMNS 308
SYSPROCEDURES 308
SYSPROCTEXT 309
SYSPROCAUTH.....309
SYSREALSTAT 310
SYSSEQAUTH 310
SYSSEQUENCES 311
SYSSYNONYMS 312
SYSTABAUTH 312
SYSTABLES_FULL 313
SYSTBLSTAT 314
SYSTIMESTAT 315
SYSTINYINTSTAT 315
SYSTRIGCOLS 316
SYSTRIGGER 316
SYSTSSTAT 317
SQL_PROPERTIES.....317
SYSSTATUSE.....318

SYSTSTZSTAT 319
SYSNVARCHARSTAT 319
SYSVIEWS 320
SYS_CHKCOL_USAGE 320
SYS_CHK_CONSTRS 320
SYS_KEYCOL_USAGE 321
SYS_REF_CONSTRS 321
SYS_TBL_CONSTRS 322

OpenEdge SQL language elements....323

OpenEdge SQL identifiers....323
Conventional identifiers....323
Delimited identifiers....324
Number formats....324
Date-time formats....325
Date formats....326
Time formats....327
Data types....328
Character data types....329
Exact numeric data types....330
Approximate numeric data types....331
Date-time data types....332
Bit string data types....333
Array data types....335
Literals....338
Relational operators....342

Data type compatibility 349

Supported ABL data types and corresponding OpenEdge SQL data types....349

OpenEdge SQL elements and statements in Backus Naur Form...351

Data types syntax in BNF....351
Data Type....351
Expressions syntax in BNF....353
Literals syntax in BNF....354
Query Expressions syntax in BNF....355
Search conditions syntax in BNF....357
Statements, DDL and DML syntax in BNF....358

Compliance with industry standards....367

Scalar functions....367
SQL-92 DDL and DML statements....371

Syntax for ABL attributes....375
OpenEdge SQL keywords for ABL table attributes....375

Part II: JDBC Reference....379

Java class reference....381
Java classes and methods....381
setParam....381
makeNULL....382
execute....382
rowCount....382
open....382
close....382
fetch....382
found....383
wasNULL....383
getValue....383
set....383
insert....383
getDiagnostics....383
log....383
err....384
DhSQLException....384
Constructors....384
Parameters....384
DhSQLException.getDiagnostics....384
DhSQLException.err....386
DhSQLResultSet....386
Constructors....386
Parameters....386
Throws....386
DhSQLResultSet.insert....386
DhSQLResultSet.makeNULL....387
DhSQLResultSet.set....388
SQLCursor....389
Constructors....389
Parameters....389
Throws....389
SQLCursor.close....390
SQLCursor.fetch....390
SQLCursor.found....391
SQLCursor.getParam....392

SQLCursor.getValue....393
SQLCursor.makeNULL....395
SQLCursor.open....395
SQLCursor.registerOutParam....396
SQLCursor.rowCount....397
SQLCursor.setParam....397
SQLCursor.wasNULL....398
SQLIStatement....399
Constructors....399
Parameter....399
Throws....400
SQLIStatement.execute....400
SQLIStatement.makeNULL....401
SQLIStatement.rowCount....401
SQLIStatement.setParam....402
SQLPStatement....403
Constructors....403
Parameter....403
Throws....403
SQLPStatement.execute....404
SQLPStatement.makeNULL....404
SQLPStatement.rowCount....405
SQLPStatement.setParam....406

JDBC conformance notes....409
Supported data types....409
Return values for DatabaseMetaData....412

Part III: ODBC Reference....423

OpenEdge SQL and ODBC data types....425

SQLGetInfo....427

ODBC scalar functions....447
Scalar functions....447
String functions....448
Numeric functions....450
Date and time functions....453
System functions....455

Part IV: ESQL Reference.....457

Embedded SQL 459

ESQL elements and statements.....459
- BEGIN-END DECLARE SECTION.....460
- CLOSE 461
- CONNECT 462
- DECLARE CURSOR 464
- DESCRIBE 465
- DESCRIBE BIND VARIABLES 465
- DESCRIBE SELECT LIST 466
- DISCONNECT 467
- EXEC SQL delimiter.....470
- EXECUTE 470
- EXECUTE IMMEDIATE 471
- FETCH 472
- GET DIAGNOSTICS 474
- GET DIAGNOSTICS EXCEPTION 475
- OPEN 478
- PREPARE 479
- SET CONNECTION 481
- WHENEVER 482

ESQL elements and statements in Backus Naur Form (BNF).....484
- BEGIN-END DECLARE SECTION 484
- Host Language Type 484
- CLOSE 484
- CONNECT 484
- CONNECT STRING 484
- DECLARE CURSOR 485
- DESCRIBE BIND VARIABLES 485
- DESCRIBE SELECT LIST 485
- DISCONNECT 485
- EXEC SQL 485
- EXECUTE 485
- EXECUTE IMMEDIATE 486
- FETCH 486
- GET DIAGNOSTICS 486
- Header Info Item 486
- GET DIAGNOSTICS EXCEPTION 486
- Detail Info Item 487
- OPEN 487
- PREPARE 487

SET CONNECTION 487
SET TRANSACTION ISOLATION LEVEL 488
ISOLATION LEVEL NAME 488
WHENEVER 488
Compliance with industry standards....488

Index....491

Copyright

© 2021 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Defrag This, Deliver More Than Expected, DevReach (and design), Icenium, Inspec, Ipswitch, iMacros, Kendo UI, Kinvey, MessageWay, MOVEit, NativeChat, NativeScript, OpenEdge, Powered by Chef, Powered by Progress, Progress, Progress Software Developers Network, SequeLink, Sitefinity (and Design), Sitefinity, Sitefinity (and design), SpeedScript, Stylus Studio, Stylized Design (Arrow/3D Box logo), Styleized Design (C Chef logo), Stylized Design of Samurai, TeamPulse, Telerik, Telerik (and design), Test Studio, WebSpeed, WhatsConfigured, WhatsConnected, WhatsUp, and WS_FTP are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries.

Analytics360, AppServer, BusinessEdge, Chef Automate, Chef Compliance, Chef Desktop, Chef Habitat, Chef WorkStation, Corticon.js, Corticon Rules, Data Access, DataDirect Autonomous REST Connector, DataDirect Spy, DevCraft, Fiddler, Fiddler Everywhere, FiddlerCap, FiddlerCore, FiddlerScript, Hybrid Data Pipeline, iMail, JustAssembly, JustDecompile, JustMock, KendoReact, NativeScript Sidekick, OpenAccess, PASOE, Pro2, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, Push Jobs, SafeSpaceVR, Sitefinity Cloud, Sitefinity CMS, Sitefinity Digital Experience Cloud, Sitefinity Feather, Sitefinity Insight, Sitefinity Thunder, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, Supermarket, SupportLink, Unite UX, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

July 2021

Product version: Progress OpenEdge 12.4

Preface

Purpose

OpenEdge SQL Reference provides specific information on the OpenEdge SQL language. The reference contains information on SQL statements, functions, reserved words, error messages, data type compatibility, and the language's compliance with industry standards. The book also provides reference information on the ODBC and JDBC drivers.

For the latest documentation updates, see the OpenEdge Product Documentation Overview page on Progress Communities:

https://community.progress.com/community_groups/openedge_general/w/openedgegeneral/1329.openedge-product-documentation-overview.aspx

Audience

The audience of this book is composed of two groups:

- **Database administrators will use the book to:**
  - Create and maintain databases
  - Create, modify, and revoke user privileges
  - Tune database performance
  - Perform installation and setup of servers and clients
- **Application developers will use the book to:**
  - Manage database connections and set up data sources
  - Create database queries
  - Tune database queries
  - Develop application business logic

Organization

Part 1, SQL Reference

OpenEdge SQL statements on page 27

Describes the purpose and syntax of each OpenEdge SQL statement. A sample is provided for each statement.

OpenEdge SQL functions

Describes the purpose and syntax of each OpenEdge SQL function. A sample is provided for each function.

OpenEdge SQL reserved words

Provides a list of words that have special syntactic meaning to OpenEdge SQL and cannot be used as identifiers for constants, variables, cursors, types, tables, records, subprograms or packages.

OpenEdge SQL error messages

Provides a list of error messages generated by the various components of OpenEdge SQL.

OpenEdge SQL system limits

Provides a list of the maximum sizes for various attributes of the OpenEdge SQL database environment, and for elements of SQL queries addressed to this environment.

OpenEdge SQL system catalog tables

Provides a set of system tables for storing information about tables, columns, indexes, constraints, and privileges. This topic describes those system catalog tables.

Data type compatiblity

Addresses compatibility issues when using OpenEdge SQL and earlier versions of the database.

OpenEdge SQL language elements

Describes Standard SQL language elements that are common to OpenEdge SQL.

OpenEdge SQL elements and statements in Backus Naur Form

Presents OpenEdge SQL elements and statements in Backus Naur Form.

Compliance with industry standards on page 367

Addresses compatibility issues when using OpenEdge SQL and earlier versions of its database.

Syntax for ABL attributes on page 375

Lists and describes SQL keywords to use with statements that allow you to define ABL attributes for tables and columns.

Part 2, JDBC Reference

Java class reference

Provides information on OpenEdge SQL Java classes and methods.

JDBC conformance notes

Provides information on mapping between JDBC and other data types and return values for database metadata.

Part 3, ODBC Reference

OpenEdge SQL and ODBC data types

Shows how the OpenEdge data types are mapped to the standard ODBC data types.

SQL GetInfo

Describes return values to SQL GetInfo from the ODBC driver.

ODBC scalar functions

Lists scalar functions that ODBC supports and are available to use in SQL statements.

Part 4, ESQL Reference

Embedded SQL

Provides reference information for an ESQL interface.

For details, see the following topics:

- Examples of SQL syntax diagrams

Purpose

OpenEdge SQL Reference provides specific information on the OpenEdge SQL language. The reference contains information on SQL statements, functions, reserved words, error messages, data type compatibility, and the language's compliance with industry standards. The book also provides reference information on the ODBC and JDBC drivers.

For the latest documentation updates, see the OpenEdge Product Documentation Overview page on Progress Communities:

https://community.progress.com/community_groups/openedge_general/w/openedgegeneral/1329.openedge-product-documentation-overview.aspx

Audience

The audience of this book is composed of two groups:

- **Database administrators will use the book to:**
  - Create and maintain databases
  - Create, modify, and revoke user privileges
  - Tune database performance
  - Perform installation and setup of servers and clients
- **Application developers will use the book to:**
  - Manage database connections and set up data sources
  - Create database queries
  - Tune database queries
  - Develop application business logic

Organization

Part 1, SQL Reference

OpenEdge SQL statements on page 27

Describes the purpose and syntax of each OpenEdge SQL statement. A sample is provided for each statement.

OpenEdge SQL functions

Describes the purpose and syntax of each OpenEdge SQL function. A sample is provided for each function.

OpenEdge SQL reserved words

Provides a list of words that have special syntactic meaning to OpenEdge SQL and cannot be used as identifiers for constants, variables, cursors, types, tables, records, subprograms or packages.

OpenEdge SQL error messages

Provides a list of error messages generated by the various components of OpenEdge SQL.

OpenEdge SQL system limits

Provides a list of the maximum sizes for various attributes of the OpenEdge SQL database environment, and for elements of SQL queries addressed to this environment.

OpenEdge SQL system catalog tables

Provides a set of system tables for storing information about tables, columns, indexes, constraints, and privileges. This topic describes those system catalog tables.

Data type compatiblity

Addresses compatibility issues when using OpenEdge SQL and earlier versions of the database.

OpenEdge SQL language elements

Describes Standard SQL language elements that are common to OpenEdge SQL.

OpenEdge SQL elements and statements in Backus Naur Form

Presents OpenEdge SQL elements and statements in Backus Naur Form.

Compliance with industry standards on page 367

Addresses compatibility issues when using OpenEdge SQL and earlier versions of its database.

Syntax for ABL attributes on page 375

Lists and describes SQL keywords to use with statements that allow you to define ABL attributes for tables and columns.

Part 2, JDBC Reference

Java class reference

Provides information on OpenEdge SQL Java classes and methods.

JDBC conformance notes

Provides information on mapping between JDBC and other data types and return values for database metadata.

Part 3, ODBC Reference

OpenEdge SQL and ODBC data types

Shows how the OpenEdge data types are mapped to the standard ODBC data types.

SQL GetInfo

Describes return values to SQL GetInfo from the ODBC driver.

ODBC scalar functions

Lists scalar functions that ODBC supports and are available to use in SQL statements.

Part 4, ESQL Reference

Embedded SQL

Provides reference information for an ESQL interface.

Examples of SQL syntax diagrams

In this example, GRANT, RESOURCE, DBA, and TO are keywords. You must specify RESOURCE, DBA, or both, and at least one *user_name*. Optionally you can specify additional *user_name* items; each subsequent *user_name* must be preceded by a comma:

Syntax

```
GRANT { RESOURCE, DBA } TO user_name[, user_name]... ;
```

This excerpt from an ODBC application invokes a stored procedure using the ODBC syntax { call *procedure_name (param)* }, where braces and parentheses are part of the language:

Syntax

```
proc1( param, "{ call proc2 (param) }", param);
```

In this example, you must specify a *table_name*, *view_name*, or *synonym*, but you can choose only one. In all SQL syntax, if you specify the optional *owner_name* qualifier, there must not be a space between the period separator and *table_name*, *view_name*, or *synonym*:

Syntax

```
CREATE [ PUBLIC ] SYNONYM synonym
  FOR [ owner_name.]{table_name | view_name | synonym} ;
```

In this example, you must specify *table_name* or *view_name*:

Syntax

```
DELETE FROM [owner_name.]{table_name | view_name}
  [ WHERE search_condition] ;
```

In this example, you must include one expression *(expr)* or column position *(posn)*, and optionally you can specify the sort order as ascending (ASC) or descending (DESC). You can specify additional expressions or column positions for sorting within a sorted result set. The SQL engine orders the rows on the basis of the first *expr* or *posn*. If the values are the same, the second *expr* or *posn* is used in the ordering:

Syntax

```
ORDER BY {expr|posn}[ ASC | DESC ]
  [ , [{expr|posn}[ ASC | DESC ]]...]
```

Long SQL syntax descriptions split across lines

Some syntax descriptions are too long to fit on one line. When syntax descriptions are split across multiple lines, groups of optional and groups of required items are kept together in the required order.

In this example, `CREATE VIEW` is followed by several optional items:

Syntax

```
CREATE VIEW [owner_name.]view_name
  [ ( column_name[, column_name]... ) ]
  AS [ ( ]query_expression[ ) ][ WITH CHECK OPTION ] ;
```

SQL Reference

For details, see the following topics:

- OpenEdge SQL statements
- OpenEdge SQL functions
- OpenEdge SQL reserved words
- OpenEdge SQL error messages
- OpenEdge SQL system limits
- OpenEdge SQL system catalog tables
- OpenEdge SQL language elements
- Data type compatibility
- OpenEdge SQL elements and statements in Backus Naur Form
- Compliance with industry standards
- Syntax for ABL attributes

1

OpenEdge SQL statements

This topic provides detailed information on OpenEdge SQL statements. A description for each statement provides the following information:

- A definition of the statement
- The syntax of the statement's proper usage
- A code sample that shows how the statement works
- Any associated notes
- Authorization required in order to run the statement
- Related statements

Remember: OpenEdge SQL does not support retrying a statement after it fails. OpenEdge SQL architecture cannot continue reliable execution after an error occurs during statement execution. You should close the failing statement and execute again.

For details, see the following topics:

- ALTER DATABASE SET PRO_ENABLE_LARGE_KEYS
- ALTER DATABASE SET PRO_ENABLE_64BIT_SEQUENCES
- ALTER GROUP
- ALTER SEQUENCE
- ALTER TABLE

- ALTER TENANT
- ALTER USER
- AUDIT INSERT
- AUDIT SET
- CALL
- Column constraints
- COMMIT
- CONNECT AS CATALOG
- CREATE DOMAIN
- CREATE GROUP
- CREATE INDEX
- CREATE PROCEDURE
- CREATE SEQUENCE
- CREATE SUPER-TENANT
- CREATE SYNONYM
- CREATE TABLE
- CREATE TENANT
- CREATE TRIGGER
- CREATE USER
- CREATE VIEW
- DELETE
- DISCONNECT CATALOG
- DROP DOMAIN
- DROP GROUP
- DROP INDEX
- DROP PROCEDURE
- DROP SEQUENCE
- DROP SYNONYM
- DROP TABLE
- DROP TENANT
- DROP TRIGGER
- DROP USER
- DROP VIEW

- GRANT
- INSERT
- LOCK TABLE
- REVOKE
- ROLLBACK
- SELECT
- SET CATALOG
- SET DATABASE DEFAULT AREA
- SET PRO_CONNECT LOG
- SET PRO_CONNECT QUERY_TIMEOUT
- SET PRO_SERVER QUERY_TIMEOUT
- SET PRO_SERVER LOG
- SET ROWCOUNT
- SET SCHEMA
- SET TRANSACTION ISOLATION LEVEL
- SHOW CATALOGS
- SHOW DATABASE DEFAULT AREA
- SHOW GROUP
- SHOW PARTITION
- SHOW ROWCOUNT
- SHOW TENANT
- SHOW ENCRYPT ON
- Table constraints
- UPDATE
- UPDATE STATISTICS

ALTER DATABASE SET PRO_ENABLE_LARGE_KEYS

This statement is used to enable large index database keys of up to 2,000 bytes.

Syntax

```
ALTER DATABASE SET PRO_ENABLE_LARGE_KEYS 'Y'
```

Notes

- You must have SQL DBA privileges to enable large keys.
- Large index keys are supported for databases with 4,000 and 8,000 byte block sizes and are enabled by default.
- Large index key sizes are enabled by default. However, in Releases 10.1A and earlier, this statement must be used to enable large keys.
- If large keys are already enabled, an error message indicating such will be returned when you use the statement.

ALTER DATABASE SET PRO_ENABLE_64BIT_SEQUENCES

This statement is used to enable large index keys in a database.

Syntax

```
ALTER DATABASE SET PRO_ENABLE_64BIT_SEQUENCES 'Y'
```

Notes

- You must have SQL DBA privileges to enable 64-bit sequences.
- OpenEdge databases support sequences with 64-bit maximums for positive and negative numbers.
- This feature is enabled by default. However, in Releases 10.1A and earlier, this specific ALTER DATABASE statement must be used to enable 64-bit sequences.
- If 64-bit sequences are already enabled, an error message indicating such will be returned when you use the statement.

ALTER GROUP

The ALTER GROUP statement allows you to add existing tenants to an existing multi-tenant group that is defined for a multi-tenant table. It also allows you to drop a tenant from an existing group that is defined for a multi-tenant table.

The tenant that you want to add to the group should not be a part of the group that is being altered. The tenant that is part of a group for a particular table cannot be part of another group for the same table.

To drop a tenant from the group that you want to alter, you must ensure that the tenant is a part of that group. When a tenant is dropped from a group, all the tenant partitions for that table are created with NO SPACE.

When the tenant of a specified table joins a group, the tenant's existing partition for that table is removed. The tenant partition for the table (on which the current group being altered is defined) must have a NO SPACE attribute set.

Syntax

```
ALTER GROUP group_name {ADD | DROP} TENANT list_of_tenant_names
```

group_name

Specifies the name of the group that you want to alter.

ADD TENANT

Adds an existing tenant to an existing multi-tenant group that is defined for a multi-tenant table.

DROP TENANT

Drops an existing tenant from an existing multi-tenant group that is defined for a multi-tenant table.

list_of_tenant_names

Specifies the list of tenants that you want to add to or drop from an existing multi-tenant group.

Examples

The following example alters a group by adding tenants to it:

```
ALTER GROUP hyb-parkinglot-group ADD TENANT easy-cabs,green-cabs,city-cabs
```

The following example alters a group by dropping tenants from it:

```
ALTER GROUP hyb-parkinglot-group DROP TENANT easy-cabs,green-cabs,city-cabs
```

Authorization

To alter a group, you must have DBA privileges.

Related Statements

CREATE GROUP, DROP GROUP

ALTER SEQUENCE

The ALTER SEQUENCE statement can be used to change the current value of an existing sequence. The sequence can be in the current schema or a schema can be specified.

When altering a sequence of a multi-tenant table as a regular tenant, the FOR TENANT phrase is optional to modify its CURRVAL, and a regular tenant does not have the privileges to modify the global attributes of a sequence.

When altering the sequence of a multi-tenant table as a DBA or a super-tenant, the FOR TENANT phrase is mandatory for a DBA or a super-tenant to identify the tenant in the multi-tenant database that must be altered. A DBA or a super-tenant can alter all the sequence attributes of a multi-tenant table.

Syntax

```
ALTER SEQUENCE [schema_name] sequence_name [SET
     {CURRVAL value [ FOR TENANT tenant_name] | START WITH value |
       INCREMENT BY value | MAXVALUE value | NOMAXVALUE |
       MINVALUE value | NOMINVALUE | CYCLE | NOCYCLE }];
```

schema_name

Specifies the name of the schema name that contains the sequence. If this is not specified, OpenEdge SQL alters the sequence.

sequence_name

Specifies the sequence to be altered.

CURRVAL

Returns the current value of the sequence.

FOR TENANT *tenant_name*

Specifies the name of the tenant corresponding to the CURRVAL.

INCREMENT BY

Specifies the interval between sequence numbers. The value can be a positive or negative integer (INTEGER data type for 32-bit sequences, BIGINT datatype for 64-bit sequences), but it cannot be 0. The value range for a 32-bit sequence is -2,147,483,648 to 2,147,483,647. The value range for a 64-bit sequence is -9223372036854775808 to 9223372036854775807. The value is positive for an ascending sequence and negative for a descending sequence. The default value is 1.

START WITH

Specifies the first number in a sequence generated. The value must be greater than or equal to the MINVALUE for an ascending sequence. The value must be less than or equal to the MAXVALUE for a descending sequence. The default value for an ascending sequence is MINVALUE and MAXVALUE for a descending sequence.

MAXVALUE

Specifies the maximum value that the sequence can generate. The default value is -1 for both 32-bit and 64-bit descending sequences. The default value is 2,147,483,647 for a 32-bit ascending sequence and 9223372036854775807 for a 64-bit ascending sequence.

NOMAXVALUE

Specifies -1 as the MAXVALUE for a 32-bit descending sequence and 2,147,483,647 as the MAXVALUE for a 32-bit ascending sequence. Specifies -1 as the MAXVALUE for a 64-bit descending sequence and 9223372036854775807 as the MAXVALUE for a 64-bit ascending sequence.

MINVALUE

Specifies the minimum value that the sequence can generate. The default value is 0 for an ascending sequence. The default value is -2,147,483,6480 for a 32-bit descending sequence and -9223372036854775808 for a 64-bit sequence.

NOMINVALUE

Specifies 0 as the MINVALUE for an ascending sequence. The MINVALUE is -2,147,483,648 for a 32-bit descending sequence and -9223372036854775808 for a 64-bit sequence.

CYCLE

Indicates that the sequence will continue to generate values after attaining the MAXVALUE (for an ascending sequence)or MINVALUE (for a descending sequence).

NOCYCLE

Indicates that the sequence cannot generate more values after attaining the MAXVALUE (for an ascending sequence) or MINVALUE (for a descending sequence). The SQL sequence generator uses NOCYCLE as the default if CYCLE is not specified.

Notes

- It is possible to set only one attribute of a sequence at a time.
- Attributes START WITH, INCREMENT BY, MAXVALUE, MINVALUE and CURRVAL can take either an INTEGER or BIGINT argument, depending on whether the sequence is a 32-bit or a 64-bit sequence.

Examples

The following example modifies a sequence by specifying a maximum value:

```
ALTER SEQUENCE pub.customer
SET MAXVALUE 9000000000;
```

The following example when executed by a DBA or a super-tenant, modifies a sequence of a multi-tenant table by specifying a maximum value:

```
ALTER SEQUENCE pub.cust_num_seq SET MAXVALUE 99999;
```

The following example when executed by a DBA or a super-tenant, modifies a sequence by specifying a CURRVAL for the tenant AsiaPecificCust of a multi-tenant table:

```
ALTER SEQUENCE pub.cust_num_seq SET CURRVAL 521 FOR TENANT mtAsiaPecificCust;
```

Authorization

To alter a sequence, you must have DBA privileges or ownership of the table.

Related Statement

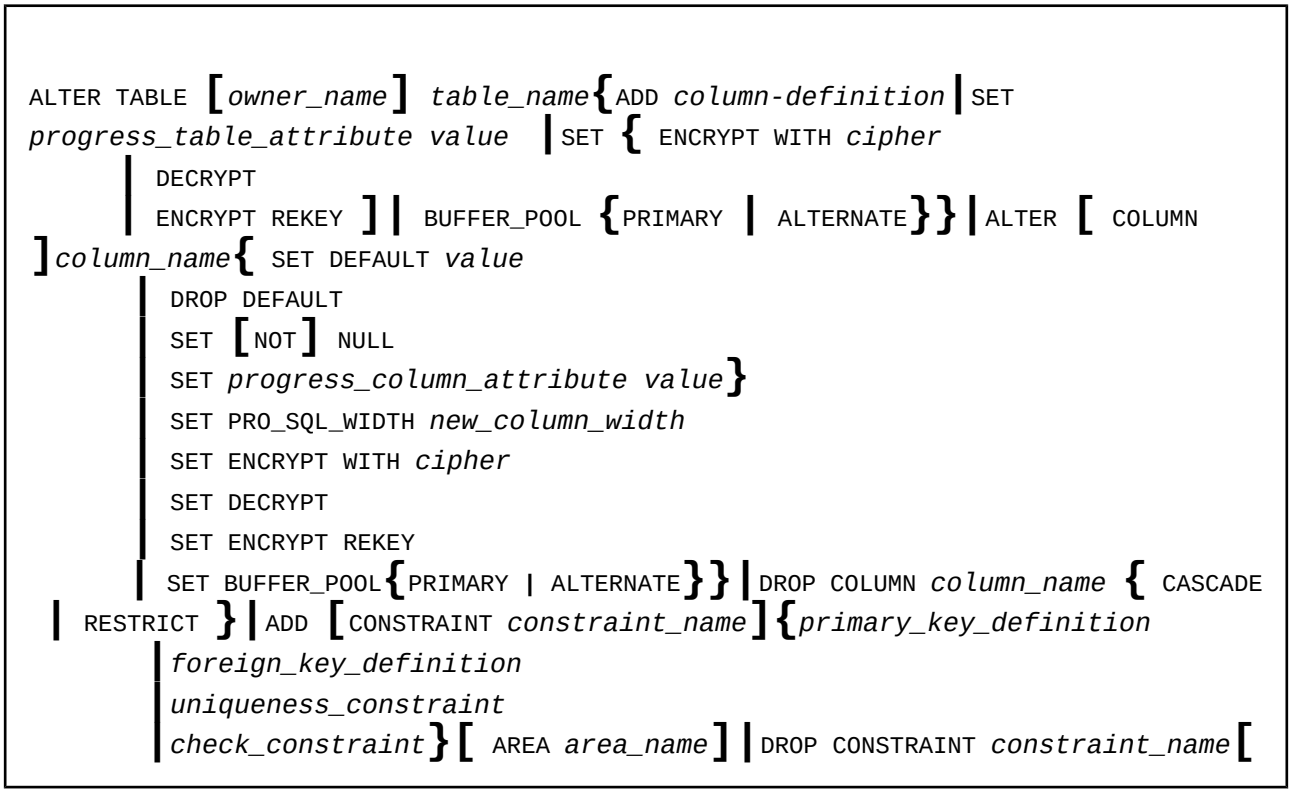
DROP SEQUENCE

ALTER TABLE

The ALTER TABLE statement can be used to:

- Change the name of a table
- Change the name of a column within a table
- Add a column to a table
- Change the width of a column
- Set (ABL) Advanced Business Language table, column and index attributes
- Convert a table to a multi-tenant table
- Add or drop LOB columns to or from a multi-tenant table
- Migrate data of an unpartitioned table to a partitioned table
- Add or drop partitions to or from a partitioned table
- Split a composite or regular (non-composite) partition into one or more partitions
- Mark partitions as read-only or read-write partitions
- Mark a table as a partitioned table without partition schema definition and then, define partition policies for it

Syntax



```
 CASCADE | RESTRICT] |ALTER INDEX index_name {SET progress_index_attribute
value
       |SET ENCRYPT WITH cipher
       |SET DECRYPT
       |SET ENCRYPT REKEY
      |SET BUFFER_POOL{PRIMARY|ALTERNATE}} |RENAME {table_name TO new_table_name

              |COLUMN column_name TO new_column_name
              |INDEX index_name TO new_index_name }}
   [ SET MULTI_TENANT
   [ FOR TENANT {[owner_name]tenant_name_1|DEFAULT}
        [ USING TABLE AREA table_area_name]
        [ USING INDEX AREA index_area_name]
        [ USING LOB AREA lob_area_name]
   ]...

   [ FOR TENANT {[owner_name]tenant_name_2|DEFAULT} USING NO SPACE ]];

ALTER TABLE ...[partition_definition ]
    USING INDEX index_name [ , index_name, ...]]
    <all other ALTER TABLE options>;

ALTER TABLE table_name
   SET READ_ONLY | READ_WRITE FOR PARTITION
       partition_name_list
```

Parameters

owner_name

Specifies the name of the schema that qualifies the table.

table_name

Specifies the name of the table.

column_definition

Defines a column. The syntax for *column_definition* is given below:

```
column_namedata_type[ COLLATE case_insensitive|case_sensitive]
  [ DEFAULT { literal | NULL | SYSDATE | SYSTIME | SYSTIMESTAMP }]
  [column_constraint[column_constraint , ... ]]
  [progress_column_attribute_keyword value], ...]
```

column_namedata_type

Names a column and associates a data type to it. The name of the column must be different from other column names already defined in the table. The *data_type* must be supported by OpenEdge.

When a table contains more than one column, a comma separator must be used after each *column_definition*, except for the final *column_definition*.

COLLATE

Indicates the case sensitivity of the column. The default value for the COLLATE element is *case_sensitive*. A collation name can be upto 32 characters long and can contain special characters like *@* or *-*. If the collation name contains special characters, the name should be entered in quotation marks similar to a SQL delimited identifier. For example *sample-one@123*. For more information on collations, refer to the Collations and collation tables section in Chapter 3 of *Internationalize ABL Applications*.

case_insensitive

Indicates that the column is *case_insensitive*. The word *case_insensitive* itself cannot be used as a valid input. The value for the *case_insensitive* clause can only be _I, I, or the default database collation with the suffix _I (For example: COLLATE_I, COLLATE I, or COLLATE BASIC_I).

case_sensitive

Indicates that the column is *case_sensitive*. The word *case_sensitive* itself cannot be used as a valid input. The value for the *case_sensitive* clause can only be _S, S, or the default database collation with the suffix _S (For example: COLLATE_S, COLLATE S, or COLLATE BASIC_S).

DEFAULT

Specifies an explicit default value for a column. The column takes on the default value if an INSERT statement does not include a value for the column. If a column definition omits the DEFAULT clause, NULL value gets assigned.

The DEFAULT clause accepts the arguments shown in the table below:

Argument	Description
literal	An integer, numeric, or a string constant.
NULL	A null value.
SYSDATE	Displays the current date. Valid only for columns defined with the DATE data type. SYSDATE is equivalent to the Progress default keyword TODAY. The argument represents a DATE value.
SYSTIME	Displays the current time. The argument represents a TIME value.
SYSTIMESTAMP	Displays the current date and time. The argument represents a TIMESTAMP value.

column_constraint

Specifies a constraint that is applied when you insert or update a value in the associated column.

progress_column_attribute_keyword value

ABL column attribute keyword and value.

The syntax used to define a LOB column is shown below:

```
{ LVARCHAR | CLOB | LVARBINARY | BLOB }
   [( length )]
   [ AREA area_name]
   [ ENCRYPT WITH cipher]
   [BUFFER_POOL{PRIMARY|ALTERNATE}]
```

progress_table_attribute_keyword value

ABL table attribute keyword and value.

PRO_SQL_WIDTH *new_column_width*

Specifies a width for the column of a table.

ENCRYPT WITH *cipher*

Allows the table to be encrypted by designating an appropriate cipher.

BUFFER_POOL {PRIMARY | ALTERNATE}

Allows the table to be assigned to a primary or alternate buffer pool.

MULTI_TENANT

Defines the table as a multi-tenant table. The MULTI_TENANT phrase can occur only after the *column_definition* and the *progress_table_attribute_keyword*.

FOR TENANT *tenant_name*

Specifies the name of a tenant. The FOR TENANT phrase can be repeated for as many existing tenants as required.

TABLE AREA *area_name*

Is used to over-ride the default storage areas associated with the particular tenant. This area is used for database space allocation during the ALTER TABLE statement execution. Specifying the *TABLE AREA* is optional.

Specifies the name of the default storage area allocated for the tenant. You can configure the default storage area using following command:

```
proutil db-name -C setobjectdefaultarea area-name [ object-type | all ]
```

INDEX AREA index *_area_name*

Is used to over-ride the default storage areas associated with the particular tenant. This area is used for database space allocation during the ALTER TABLEstatement execution. Specifying the *INDEX AREA* is optional.

If the storage area for an index is omitted in the syntax, then any index will be allocated in the table area.

To define *INDEX AREA*, you must define the *TABLE AREA*.

LOB AREA lob *_area_name*

Is used to over-ride the default storage areas associated with the particular tenant. This area is used for database space allocation during the ALTER TABLE statement execution. Specifying the *LOB AREA* is optional.

If the storage area for a LOB is omitted in the syntax, then any LOB will be allocated in the table area.

To define the *LOB AREA*, you must define the *TABLE AREA*.

USING NO SPACE

Indicates that no space is allocated for index data storage in the table. This phrase is valid only if the tenant partition for the table is allocated NO SPACE.

Note: For a particular tenant, the table and the Index partitions must have the same allocation state.

partition_definition

Uses the following syntax:

```
[PARTITION BY {RANGE | LIST}column_name
 [PRO_DESCRIPTION desc]
 [subpartition_definition, . . .]area_spec( partition_attribute
[partition_attribute ], . . .)
```

The *partition_definition* must contain values for all partition keys. If the partition keys are x, y, and z, then all the partition definitions must specify values for columns x, y, and z.

subpartition_definition

Uses the following syntax:

```
SUBPARTITION BY {RANGE | LIST} column_name [PRO_DESCRIPTION desc]
```

column_name

Uses the following syntax:

```
simple_column_name | (simple_column_name )
```

partition_attribute

Uses the following syntax:

```
PARTITION [partition_name ] VALUES { <= | IN } (column_values)
  [area_spec ] [ PRO_DESCRIPTION desc ]
```

partition_name

Specifies the SQL identifier.

column_values

Uses the following syntax:

```
column_value [ , column_value, ...]
```

column_value

Uses a constant as its value. Each *column_value* corresponds to a column name of a partition or a subpartition, in the same order in which the column names are defined.

area spec

Specifies the default areas for all partitions defined, if not overridden by specific partition definitions. The syntax for *area spec* is:

```
( ( [ USING TABLE AREA area name] [ USING INDEX AREA area name]
    [ USING LOB AREA area name] ) | [ USING NO SPACE ] )
```

index_name

Specifies the SQL identifier. The index must be a partition aligned index.

Notes

- See Syntax for ABL attributes on page 375 for a list of ABL table, column and index attributes.
- **OE SQL command behavior for `Alter table <tbl-name> alter column <col-name> set pro_sql_width <new_width>`**

OE SQL will return the error `Invalid value for attribute "PRO_SQL_WIDTH" specified for column arr. (19164)` for `alter table <tblname> alter <colname> set pro_sql_width` where, columns are of array type (either array or vararray) and the new width is less than the minimum required value.

This behavior is applicable only for columns of array type (either array or vararray) where, the new width should be more than the minimum defined value. The minimum value can be calculated as `min_sql_width = (extent * 2 ) -1` , where, the `extent = number of elements in array`.

Example

Consider the following tables.

```
create table t1( arr varchar(10) array[3]);
```

Here, the extent is 3. So, `min_sql_width = (3 *2 ) -1 = 5` .

So, the ALTER command `alter table t1 alter column arr set pro_sql_width 3` will fail as the new value 3 is less than the minimum required value. It will fail with the error code `210157` and message `Invalid value for attribute "PRO_SQL_WIDTH" specified for column arr. (19164)` .

Also, when a user provides the new_width in the `alter command`, the OE SQL engine rounds up the new width to the nearest number divisible by extent.

If `new_width` is 3, it will remain as 3.

If `new_width` is `4` , it will be rounded up to 6.

So, the ALTER command `alter table t1 alter column arr set pro_sql_width 4` will pass, even though the new value is less than minimum required value, because it is rounded up to 6.

- The `ALTER TABLE ALTER INDEX` statement can use two index attributes, `PRO_DESCRIPTION` and `PRO_ACTIVE`. The `PRO_DESCRIPTION` attribute enables the index definition to accept free-form text in the same manner as ABL. The `PRO_ACTIVE` attribute takes only n as an argument, thereby changing the status of an index from active to inactive. Changing an status of an index to inactive is an action that must be performed offline. For a description of the `PRO_DESCRIPTION` and `PRO_ACTIVE` attributes, see Syntax for ABL attributes on page 375.
- Table columns defined by OpenEdge SQL have default format values identical to those created by the Data Dictionary.
- For details on using the `ALTER TABLE ADD COLUMN` statement to designate objects for buffer pool assignments, including an alternate buffer pool, see *Manage the OpenEdge Database*.
- To enable transparent data encryption using the `ALTER TABLE` statement, see *Learn about Security and Auditing*.

Examples

Changing the name of a table

In the following example, the `ALTER TABLE` statement is used to change the name of a table from `customer` to `Customers`:

```
ALTER TABLE customer RENAME TO Customers;
```

Changing the name of a column within a table

In this example, the ALTER TABLE statement is used to change the name of a column within a table and the column named Address changes to Street:

```
ALTER TABLE customer RENAME Address TO Street;
```

Adding a column

In this example, table customer adds the column Region:

```
ALTER TABLE customer ADD COLUMN Region;
```

Modifying a column

In this example, table customer changes an existing 32-bit INTEGER column into a 64-bit BIGINT column:

```
ALTER TABLE OrderLine ALTER COLUMN Qty SET PRO_DATA_TYPE BIGINT;
```

Once the above statement is executed, the column will appear as a BIGINT column both internally and to applications.

The above statement executed against a column that is not a 32-bit integer results in a syntax error.

Adding an ABL description to a table and change the ABL default data access index of the table

In this example, ALTER TABLE adds an ABL description to a table and changes the ABL default data access index of the table:

```
ALTER TABLE pub.customer SET PRO_DESCRIPTION 'Sports 2020 Customers';
ALTER TABLE pub.customer SET PRO_DEFAULT_INDEX CustNumIdx;
```

Renaming an index

In this example, ALTER TABLE RENAME INDEX is used to change an index named CustNum to CustomerNumberIndex:

```
ALTER TABLE Customers RENAME INDEX CustNum to CustomerNumberIndex;
```

The ALTER TABLE statement enables you to change the names of tables or columns or to add columns while your database is online servicing other requests. Other changes performed by ALTER TABLE must occur offline.

Marking existing partitions as read-only partitions

```
ALTER TABLE Pub.tpcustomer SET READ_ONLY FOR PARTITION
NoholdDKP15500, NoholdBBB15500, HoldSRS10000;
```

Marking existing partitions as read-write partitions

```
ALTER TABLE Pub.tpcustomer SET READ_WRITE FOR PARTITION
NoholdDKP15500, NoholdBBB15500, HoldSRS10000;
```

See also

Defining LOB columns on page 42
Converting a table to a multi-tenant table on page 44
Repairing a CRC mismatch error on page 45
Migrating data from existing unpartitioned tables to partitioned tables on page 46
ALTER TABLE syntax to add or drop partitions on page 62

Defining LOB columns

The below syntax is used to define an *LOB* column in `ALTER TABLE ADD COLUMN` statement:

```
{ LVARCHAR | CLOB | LVARBINARY | BLOB }[ ( length ) ][ AREA area_name ][ ENCRYPT
 WITH cipher ][BUFFER_POOL{PRIMARY|ALTERNATE}]
```

ALTER TABLE RENAME COLUMN

The main features of `ALTER TABLE RENAME COLUMN` statement are:

- `ALTER TABLE RENAME COLUMN` is an online and offline operation.
- `ALTER TABLE RENAME COLUMN` is an offline operation for instances where:
  - Table is in PUB schema.
  - Database is enabled for OpenEdge Replication.
- `ALTER TABLE RENAME COLUMN` is an online operation for tables defined in non-PUB schema

Impact

When `ALTER TABLE RENAME COLUMN` is performed, the following are impacted:

- **OpenEdge Management** — OpenEdge Management does not display table definitions for non-PUB schema except for tables that are Change Data Capture (CDC) enabled. When CDC operations are performed in concurrent transaction with `ALTER TABLE RENAME COLUMN`, OpenEdge Management may return an error because of lock conflict.

  Even without lock conflicts, if you change a column name between the time OpenEdge Management reads the definition and uses the definition via ABL, creating the required CDC policy fails with `Column not found` error.

- **Progress Developer Studio for OpenEdge** — Progress Developer Studio for OpenEdge can fetch table and column definitions, and display them to perform database operations.

When you perform online `ALTER TABLE RENAME COLUMN`, in concurrent transaction, and when Progress Developer Studio for OpenEdge fetch the data using old column name, then it might return `Column not found` error.

- **Database utilities** — When you perform online `ALTER TABLE RENAME COLUMN` in concurrent operation with database utilities or index utilities (IDXcheck,IDXfix), they might result in lock conflict.

ALTER TABLE RENAME INDEX

The main features of `ALTER TABLE RENAME INDEX` statement are:

- `ALTER TABLE RENAME INDEX` is an online and offline operation.
- `ALTER TABLE RENAME INDEX` is an offline operation for instances where:
  - Table is in PUB schema.
  - Database is enabled for OpenEdge Replication.
- `ALTER TABLE RENAME INDEX` is an online operation for tables defined in a non-PUB schema.

Impact

When `ALTER TABLE RENAME INDEX` is performed, the following are impacted:

- **Progress Developer Studio for OpenEdge** — Progress Developer Studio for OpenEdge can fetch table and index definitions, and display them to perform database operations.

  When you perform online `ALTER TABLE RENAME INDEX`, in concurrent transaction, and when Progress Developer Studio for OpenEdge fetch the data using old index name, then it might return `Index not found` error.

- **Database utilities** — When you perform online `ALTER TABLE RENAME INDEX` in concurrent operation with database utilities or index utilities (IDXcheck,IDXfix), they might result in lock conflict.

ALTER TABLE DROP COLUMN

Deletes a Column.

It is now possible to perform an ‘ALTER TABLE DROP COLUMN’ on a non-PUB schema without disconnecting concurrent users/clients. Drop column can be performed online while other users/clients can access the OpenEdge database objects.

The main features of the `ALTER TABLE DROP COLUMN` statement are:

- `ALTER TABLE DROP COLUMN` is an online and offline operation.
- `ALTER TABLE DROP COLUMN` is an offline operation for instances, where:
  - The table is in PUB schema.
  - Database is enabled for OpenEdge Replication.
  - Column being dropped is of LOB type i.e. CLOB (Character Large Object) or BLOB (Binary Large Object) data types.
  - Column being dropped is a partition key column.

Impact

When ALTER TABLE DROP COLUMN is performed, the following are impacted:

- **OpenEdge Management** — OpenEdge Management does not display table definitions for non-PUB schema except for tables that are Change Data Capture (CDC) enabled. When CDC operations are performed in concurrent transaction with ALTER TABLE DROP COLUMN, OpenEdge Management may return an error because of lock conflict.

  Even without lock conflicts, if you change a column name between the time OpenEdge Management reads the definition and uses the definition via ABL, creating the required CDC policy fails with Column not found error.

- **Progress Developer Studio for OpenEdge** — Progress Developer Studio for OpenEdge can fetch table and column definitions, and display them to perform database operations.

  When you perform online ALTER TABLE DROP COLUMN, in concurrent transaction, and when Progress Developer Studio for OpenEdge fetch the data using old column name, then it might return Column not found error.

- **Database utilities** — When you perform online ALTER TABLE DROP COLUMN in concurrent operation with database utilities or index utilities (IDXcheck,IDXfix), they might result in lock conflict.

Converting a table to a multi-tenant table

Syntax

The syntax used to convert a table to a multi-tenant table using the ALTER TABLE statement is given below:

```
[ SET MULTI_TENANT
    [ FOR TENANT { [ owner_name ] tenant_name_1 | DEFAULT }
        [ USING TABLE AREA table_area_name ]
        [ USING INDEX AREA index_area_name ]
        [ USING LOB AREA lob_area_name ]
    ]...
    [ FOR TENANT { [ owner_name ] tenant_name_2 | DEFAULT } USING NO SPACE ] ]
```

Examples

Altering a table to be a multi-tenant table with default space allocations

The following example changes the pub.mtcustomer table to a multi-tenant table with default space allocations:

```
ALTER TABLE pub.mtcustomer SET MULTI_TENANT;
```

Altering a table to be a multi-tenant table with overriding space allocations

The following example alters the pub.mtcustomer table to be a multi-tenant table with overriding space allocations:

```
ALTER TABLE pub.mtcustomer  SET MULTI_TENANT
FOR TENANT "Consolidated_Freightways"
               USING TABLE AREA "MMM Data area"
               USING INDEX AREA "MM CF Fin idx area"
FOR TENANT Mega_Media_Networks
               USING LOB AREA "MMM pix area" ;
 FOR TENANT DEFAULT USING NO SPACE;
```

In general, the above example describes how to convert a regular table to a multi-tenant table. When a regular table is converted to a multi-tenant table, the table data is moved to the default partition. To move the data from the default partition to the tenant partitions, see the INSERT statement.

Repairing a CRC mismatch error

The following special syntax is for Database Administrators who handle problems while migrating database tables from an older release database to a newer release database. In some cases, there may be a CRC mismatch error (when dumping and loading data using `BINARY DUMP/LOAD`.).

Syntax

```
ALTER TABLE table_name SET PRO_FIELD_INFO PRO_SCHEMA_REPAIR
```

Note: The `PRO_FIELD_INFO` attribute indicates that the incorrect values in the`_Field` table, corresponding to the given table name, need to be updated. The `PRO_SCHEMA_REPAIR` attribute provides additional authorization to DBAs that is required to perform the necessary updates to CRC specific rows in system tables.

Using the above statement for a particular table automatically repairs all the known schema errors in the table that are causing the CRC mismatch. If the error persists even after running this statement, report the problem to Progress Customer Support with the `_Field` system table information in the old and new databases.

Execution of the above statement sets `_DB-lang` to its default value (0) and updates the following data type names as shown below:

- `TIMESTAMP` to `DATETIME`
- `LVARBINARY` to `BLOB`
- `LVARCHAR` to `CLOB`

Executing the above statement also sets the case sensitivity value of the non-character type columns to 0.

Example

The following example repairs the CRC mismatch error:

```
ALTER TABLE Customer SET PRO_FIELD_INFO PRO_SCHEMA_REPAIR;
```

In the above example, the `Customer` table belongs to an old database. When you execute the above statement, the inconsistent field information is corrected and the table is updated with the new CRC.

Migrating data from existing unpartitioned tables to partitioned tables

You can use the ALTER TABLE statement to migrate data of an unpartitioned table to a partitioned table. The ALTER TABLE statement for table partitioning functions similarly as the CREATE TABLE statement, except that the ALTER TABLE statement creates only the partition definition and points them to the current table partition (composite initial partition).

Note: The composite initial partition contains the data of a table before it is partitioned and all the partition-aligned indexes specified in the USING INDEX clause are known as composite indexes; Partition-aligned indexes are indexes that use partitioned key columns as the leading prefix components.

Data access from the new partition definition is automatically redirected to the composite initial partition, which is then gradually split into new actual partitions using a database utility.

Since all the data resides in the composite partition during the initial migration, all the partitions must be either read-only partitions or read-write partitions. If all the partitions are marked as read-only partitions, then the composite partition is marked as a read-only partition. If all the partitions are marked as read-write partitions, then the composite partition is marked as a read-write partition. An error occurs if a composite partition is created with both read-only partitions and read-write partitions.

Note: A NO SPACE partition cannot be marked as a read-only partition.

Migrating data of an unpartitioned table to a partitioned table involves the following steps:

1. Enabling a table for partitioning on page 47
2. Marking partitions as split targets on page 52
3. Moving the physical data from the composite initial partition to its respective partitions on page 56

Note: If the table does not contain any data (if the number of rows in the table is zero), then this first step is sufficient to migrate data from an unpartitioned table to a partitioned table. This first step creates all the necessary partitions thereby, avoiding the need to prepare split targets and executing the database utility.

Examples

Marking a composite partition as a read-only partition while migrating data of an unpartitioned table to a partitioned table

```
ALTER TABLE Pub.tpcustomer PARTITION BY LIST CustID
USING TABLE AREA "Tenant_1_table_Area"
(
    PARTITION tpcustomer_PRTN1 VALUES IN (1001) READ_ONLY,
    PARTITION tpcustomer_PRTN2 VALUES IN (1002) READ_ONLY,
    PARTITION tpcustomer_PRTN3 VALUES IN (1003) READ_ONLY,
    PARTITION tpcustomer_PRTN3 VALUES IN (1004) READ_ONLY,
)
    USING INDEX tpcustomer_IDX3;
```

Marking a composite partition as a read-write partition while migrating data of an unpartitioned table to a partitioned table

```
ALTER TABLE Pub.tpcustomer PARTITION BY LIST CustID
USING TABLE AREA "Tenant_1_table_Area"
(
    PARTITION tpcustomer_PRTN1 VALUES IN (1001) READ_WRITE,
    PARTITION tpcustomer_PRTN2 VALUES IN (1002) READ_WRITE,
    PARTITION tpcustomer_PRTN3 VALUES IN (1003) READ_WRITE,
    PARTITION tpcustomer_PRTN3 VALUES IN (1004) READ_WRITE,
)
    USING INDEX tpcustomer_IDX3;
```

Note: An error occurs if the composite partition is created with both read-only partitions and read-write partitions.

Marking a composite partition as a read-only partition

The following example illustrates an alternative syntax for marking a composite partition as a read-only partition:

```
ALTER TABLE Pub.tpcustomer SET READ_ONLY FOR PARTITION PRO_INITIAL;
```

Marking a composite partition as a read-write partition

The following example illustrates an alternative syntax for marking a composite partition as a read-write partition:

```
ALTER TABLE Pub.tpcustomer SET READ_WRITE FOR PARTITION PRO_INITIAL;
```

Enabling a table for partitioning

The following ALTER TABLE syntax defines the partition of a table. Here, the actual data continues to reside in the composite initial partition and the newly added partitions point to the composite initial partition. The data can be accessed normally after the execution of this syntax.

Syntax

```
ALTER TABLE ...[ partition_definition ]
  USING INDEX index_name [ , index_name, ...]
```

Parameters

partition_definition

Uses the following syntax:

```
PARTITION BY {RANGE|LIST} column_name
  [PRO_DESCRIPTION desc] [subpartition_definition,...] area_spec
(partition_attribute [partition_attribute ], ...)
```

The *partition_definition* must contain values for all partition keys. If the partition keys are x, y, and z, then all the partition definitions must specify values for columns x, y, and z.

subpartition_definition

Uses the following syntax:

```
SUBPARTITION BY {RANGE|LIST} column_name [PRO_DESCRIPTION desc]
```

column_name

Uses the following syntax:

```
simple_column_name | (simple_column_name )
```

partition_attribute

Uses the following syntax:

```
PARTITION [partition_name ] VALUES { <= | IN } (column_values)
[area_spec ] [ PRO_DESCRIPTION desc ]
```

partition_name

Specifies the SQL identifier.

column_values

Uses the following syntax:

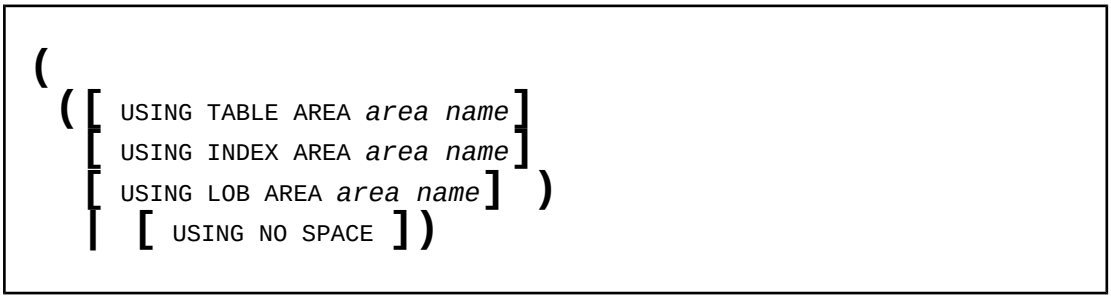
```
column_value [ , column_value, ...]
```

column_value

Uses a constant as its value. Each *column_value* corresponds to a column name of a partition or a subpartition, in the same order in which the column names are defined.

area spec

Specifies the default areas for all partitions defined, if not overridden by specific partition definitions. The syntax for *area spec* is:

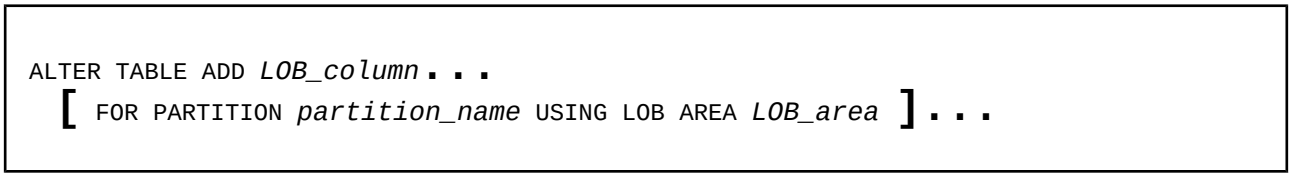


index_name

Specifies the SQL identifier. The index must be a partition aligned index.

Specifying partition specific areas on LOB columns

Use the following syntax to specify partition specific areas on LOB columns:



Notes

- The ALTER TABLE statement for table partitioning is an online operation.
- Data of an unpartitioned table can be migrated to a partitioned table only if the table and all its indexes are in the type II area.
- Partition names are optional, unique across a database, and have the lexical properties of a table name; if not specified, system generated partition names are used.
- The maximum number of partitions per table is 32,767.
- A partition can be defined without any allocated data storage.
- Partition keys must be literal constants and must be enclosed in parentheses for LIST partitions. For RANGE partitions, the parentheses are optional.
- Table partitioning must be enabled for a database to use the table partitioning syntax with the ALTER TABLE statement.
- All index supported data types can be used as partition keys.
- The area definition at the partition level always takes priority among the area definitions provided at multiples levels.
- While migrating data from an unpartitioned table to a partitioned table, the ALTER TABLE statement validates the data in the existing table based on whether the defined partitions meet the following requirements:

  - Only one RANGE partition definition is allowed per table such that the single RANGE partition defined has a value greater than or equal to the column's maximum value. If the table contains subpartitions, then only one RANGE partition is allowed per list.
  - Any number of LIST partitions can be defined as long as all the data in the column belongs to the partitions defined. For any row that does not belong to the defined partitioned, the ALTER TABLE statement returns an error.

- For a subpartition, all the leading subpartition levels must define LIST partitions. The last subpartition level can either be a RANGE partition or a LIST partition.
- There can be up to 15 levels of subpartitioning.
- Partitioned key columns must not be repeated.
- If the partition definition has a RANGE column, then the <= clause must be used to specify the partition key values, otherwise use the IN clause.
- For RANGE partitions having the same key values, the values must be specified sequentially in an ascending order.
- The following semantics apply when using the FOR PARTITION clause in the ALTER TABLE statement to add LOB columns:
  - The table specified in the statements must be a partitioned table
  - The partition specified in the FOR PARTITION clause must be an existing partition in the table
  - Partitions can be specified in any order
  - The specified area must be present in the table and in the type II area
  - Specification of any area other than the LOB area (for example, table area) results in an error

Examples

Using ALTER TABLE with partitioned tables

The following example illustrates how to use the ALTER TABLE statement with partitioned tables.

The below statement creates a table with two partition aligned indexes: IDX1 and IDX2 and an index IDX3 that is not partition aligned.

```
CREATE TABLE Pub.tpcustomer
(    Custid  int,
    Custname varchar (50),
    join_date date,
    salary int
);
CREATE INDEX IDX1 on Pub.tpcustomer (custid);
CREATE INDEX IDX2 on Pub.tpcustomer (custid,salary);
CREATE INDEX IDX3 on Pub.tpcustomer (salary);
```

The below ALTER TABLE statement converts the above table to a partitioned table by adding one data partition definition and specifies IDX1 as the local composite index. IDX2 and IDX3 are marked as global indexes. If the data in Pub.tpcustomer contains custid greater than 100000, then the statement returns an error.

```
ALTER TABLE Pub.tpcustomer
 PARTITION BY RANGE Custid
  USING TABLE AREA "custtab_area"
  USING INDEX AREA "custidx_area" (  PARTITION p3 VALUES <= (100000) )
  USING INDEX IDX1;
```

Using ALTER TABLE with subpartitioned tables

The following example illustrates how to use the ALTER TABLE statement with subpartitioned tables.

In the below example, Pub.sub_customer_list is LIST partitioned by zipcode and custid.

```
CREATE TABLE Pub.sub_customer_list
  (   Custid  int,
    Custname varchar(50),
    Zipcode int
  )
 AREA "TP area";


CREATE INDEX sub_customer_list_list_idxlc1
  ON pub.sub_customer_list_list (Zipcode, Custid, Custname)
  AREA "TP idx area";

CREATE INDEX sub_customer_list_list_idxlc2
 ON pub.sub_customer_list_list (Zipcode, Custid)
 AREA "TP idx area";

CREATE INDEX sub_customer_list_list_idxg1

 ON pub.sub_customer_list (Custid)
 AREA "TP idx area";

ALTER TABLE pub.sub_customer_list
 PARTITION BY LIST Zipcode
 SUBPARTITION BY LIST Custid
 USING TABLE AREA "TP Area"
  (
  PARTITION sub_customer_lista1 VALUES IN (28019,801) ,
  PARTITION sub_customer_lista2 VALUES IN (28019,802) ,
  PARTITION sub_customer_listab VALUES IN (28019,803) ,
  PARTITION sub_customer_listb1 VALUES IN (28028,1804) ,
  PARTITION sub_customer_listb2 VALUES IN (28028,1805) ,
  PARTITION sub_customer_listb3 VALUES IN (28028,1806) ,
  PARTITION sub_customer_listc1 VALUES IN (28039,2807),
  PARTITION sub_customer_listc2 VALUES IN (28039,2808),
  PARTITION sub_customer_listc3 VALUES IN (28039,2809)  )
  USING INDEX  sub_customer_list_idxlc1, sub_customer_list_idxlc2;
```

Marking partitions as split targets

The following ALTER TABLE split partition syntax prepares an existing partition to be split into one or more partitions.

Syntax

```
ALTER TABLE table_name
  [PREPARE FOR SPLIT {partition name | PRO_INITIAL }
    [{range_partition_split | list_partition_split ]]
   <all other ALTER TABLE options>
```

Parameters

partition name

Refers to regular, non-composite RANGE partition name to be split. It reports an error if a LIST partition name is specified.

PRO_INITIAL

Refers to the composite initial partition of the migrated table.

range_partition_split

Uses the following syntax:

```
TARGET{PARTITION | PARTITIONS [partition name ] [ RO_RW_ATTRIBUTE ]area
spec[, ...]
```

column_values

Uses the following syntax:

```
column_value [ , column_value, ...]
```

column_value

Uses a constant as its value. Each *column_value* corresponds to a column name of a partition or a subpartition, in the same order in which the column names are defined.

RO_RW_ATTRIBUTE

Uses the following syntax and marks the partition as a read-only partition or a read-write partition:

```
READ_ONLY | READ_WRITE
```

If RO_RW_Attribute is not specified, then by default, the partition is marked as a read-write partition.

list_partition_split

Uses the following syntax:

```
TARGET {PARTITION | PARTITIONS } partition_name
   [ RO RW ATTRIBUTE ][, ...]
```

RO_RW_ATTRIBUTE

Uses the following syntax and marks the partition as a read-only partition or a read-write partition:

```
READ_ONLY | READ_WRITE
```

If RO_RW_Attribute is not specified, then by default, the partition is marked as a read-write partition.

Splitting a regular, non-composite LIST partition is not supported. Only a RANGE partition can be split into one or more partitions. After splitting composite RANGE partitions into several partitions, the remaining RANGE partitions can be marked as split targets by executing the following syntax:

```
ALTER TABLE table_name PREPARE FOR SPLIT PRO_INITIAL;
```

Notes

The following semantics apply when using ALTER TABLE split initial partition (PRO_INITIAL) syntax:

- The table specified in the syntax must be a partitioned table
- While executing this command to split an initial composite partition (PRO_INITIAL), the table specified must be a migrated table.
- Split PRO_INITIAL marks all partitions referring to initial composite partition as split targets.
- The partition specified in the syntax cannot be accessed after execution, until the data is moved using the PROUTIL utility. However, it can be accessed by queries through the global index scan operation.
- The execution of this syntax creates the necessary target storage objects and their respective index partitions.
- In case of RANGE partitions, only one partition definition refers to the composite initial partition per a set of same leading LIST partition keys.
- All partition definitions referring to the composite initial partition are marked as split targets when the initial partition is split without new partition definitions (when neither *range_partition_split* nor *list_partition_split* is specified).
- The following semantics apply using ALTER TABLE split partition syntax where the initial partition is split with new partition definitions, with *range_partition_split* specified:
  - All the new partitions added are marked as split targets
  - The command can be used only with RANGE partitions
  - The new partition definitions must be within the boundaries of a composite partition

- If the split source specified is a non-composite RANGE partition, then the source partition name must be specified using the PARTITION option of split utility.

- The following semantics apply using ALTER TABLE split partition syntax where the initial partition is split with specified target partitions, with *list_partition_split* specified:
  - All the partitions specified in the clause are marked as split targets.
  - If the partitions specified in the clause are already marked as split targets, the SQL engine returns an error.
  - Data in the specified partitions, once marked as split targets, is not accessible. It can be accessed from all other partitions that are not marked as split targets.
  - PARTITION and PARTITIONS clauses can be used interchangeably.
  - Execution of the above syntax results in an error if the partition that is marked as the split target is accessed unless the data is moved using the split utility.
  - Execution of the above syntax creates necessary storage objects and the respective index partitions for target partitions.

Examples

Marking partitions as read-only and read-write partitions while splitting non-composite RANGE partitions

```
ALTER TABLE Pub.tpcustomer
PREPARE FOR SPLIT NoholdDKP15500
(
  PARTITION NoholdDKP10000 VALUES <= (0, 'DKP', 10000) READ_ONLY,
  PARTITION NoholdDKP11000 VALUES <= (0, 'DKP', 11000),
  PARTITION NoholdDKP12000 VALUES <= (0, 'DKP', 12000),
  PARTITION NoholdDKP13000 VALUES <= (0, 'DKP', 13000) READ_WRITE,
  PARTITION NoholdDKP14000 VALUES <= (0, 'DKP', 14000) READ_WRITE,
  PARTITION NoholdDKP15000 VALUES <= (0, 'DKP', 15000) READ_ONLY
);
```

Note: Unlike the composite partitions, non-composite partitions can have both read-only and read-write partitions for the same table.

Data movement from a read-only source partition to a read-write target partition is not allowed. In the above example, the source partition NoHoldDKP15500 is a read-only partition, and since some of the split targets are marked as read-write partitions, the PROUTIL utility returns an error during the data movement.

Marking partitions as read-only and read-write partitions while splitting composite partitions

```
ALTER TABLE Pub.tpcustomer
  PREPARE FOR SPLIT PRO_INITIAL TARGET PARTITIONS
   tpcustomer_partn1 READ_ONLY,
   tpcustomer_partn2 READ_ONLY,
   tpcustomer_partn4 READ_WRITE,
   tpcustomer_partn5 READ_ONLY;
```

Note: Data movement from a read-only source partition to a read-write target partition is not allowed.

Moving the physical data from the composite initial partition to its respective partitions

After the composite or regular (non-composite) partitions are split into several partitions and marked as split targets, the data residing in the composite initial partition or the source RANGE partition can be moved to the respective partitions using the PROUTIL utility.

After the data is migrated, the process of converting an unpartitioned table to a partitioned table or splitting an existing RANGE partition to one or more partitions is complete.

Note: Running the PROUTIL split utility with the composite initial option moves the data from the partitions that are added and marked as split targets, leaving the remaining data in the composite partition.

Examples

Migrating an unpartitioned table to a partitioned table

The ALTER TABLE statement in the following example migrates data of an unpartitioned table to a partitioned table. The table in the example contains three columns and three indexes (out of which two are partition aligned):

```
CREATE TABLE Pub.tpcustomer
 (
   Custid int,
   Custname varchar (50),
   Zipcode int
 )
   AREA "custtab_area";

CREATE INDEX Customer_idx_local1 on Pub.tpcustomer
 (zipcode)
  AREA "custidx_area";

CREATE INDEX Customer_idx_local2 on Pub.tpcustomer
 (zipcode,custid)
  AREA "custidx_area";

CREATE INDEX Customer_idx_globa1 on Pub.tpcustomer (Custid) AREA "custidx_area";

INSERT INTO Pub.tpcustomer VALUES (801, 'cust1', 28019);

INSERT INTO Pub.tpcustomer VALUES (802, 'cust2', 28019);

INSERT INTO Pub.tpcustomer VALUES (803, 'cust3', 28019);

INSERT INTO Pub.tpcustomer VALUES (1804, 'cust4',28028);

INSERT INTO Pub.tpcustomer VALUES (1805, 'cust5',28028);

INSERT INTO Pub.tpcustomer VALUES (1806, 'cust6',28028);

INSERT INTO Pub.tpcustomer VALUES (2807, 'cust7',28039);

INSERT INTO Pub.tpcustomer VALUES (2808, 'cust8',28039);

INSERT INTO Pub.tpcustomer VALUES (2809, 'cust9',28039);

ALTER TABLE Pub.tpcustomer

  PARTITION BY LIST zipcode

  USING TABLE AREA "custtab_area"
  USING INDEX AREA "custidx_area"
  (
  PARTITION ZIP_28019 VALUES IN (28019),
  PARTITION ZIP_28029 VALUES IN (28028),
  PARTITION ZIP_28039 VALUES IN (28039)
  )
  USING INDEX Customer_idx_local1, Customer_idx_local2;
```

The above ALTER TABLE statement creates partition definition of the table with the data in the composite initial partition. All the three newly added partitions point to the composite initial partition. The data can be accessed normally.

The following ALTER TABLE SPLIT statement marks all the three partitions as ready to be split. After the statement is executed, the data access to these partitions results in an error.

```
ALTER TABLE Pub.tpcustomer PREPARE FOR SPLIT PRO_INITIAL;
```

The following statement moves the data to its respective partitions. At this stage, each partition contains three rows and the table is completely partitioned with three LIST partitions.

```
$PROUTIL testdb -C partitionmanage split table Pub.tpcustomer composite initial
useindex Customer_idx_local1
```

Migrating data of an unpartitioned table to a RANGE partitioned table

The following statement creates a table with four indexes, out of which, the first two are partition aligned.

The below statement alters the table to have a single RANGE partition:

```
CREATE TABLE Pub.tpcustomer
  (
    Custid  int,
    Custname varchar(50),
    join_date date,
    salary int
  )
   AREA "Customer_Table_Area";

CREATE INDEX tpcustomer_idxc21 on pub.tpcustomer(Custid) AREA "Customer_Table_Area";

CREATE INDEX tpcustomer_idxc22 on pub.tpcustomer(Custid, Custname) AREA
"Customer_Table_Area";

CREATE INDEX tpcustomer_idxg21 on pub.tpcustomer(join_date) AREA "Customer_Table_Area";

CREATE INDEX tpcustomer_idxg22 on pub.tpcustomer(salary) AREA "Customer_Table_Area";

INSERT INTO Pub.tpcustomer VALUES(800, 'cust 1', '01/06/2013', 5000);

INSERT INTO Pub.tpcustomer VALUES(850, 'cust 2', '01/06/2013', 5000);

INSERT INTO Pub.tpcustomer VALUES(900, 'cust 3', '01/06/2013', 5000);

INSERT INTO Pub.tpcustomer VALUES(1000, 'cust 4', '01/06/2013', 5000);

INSERT INTO Pub.tpcustomer VALUES(1200, 'cust 5', '01/06/2013', 5000);

INSERT INTO Pub.tpcustomer VALUES(1400, 'cust 6', '01/06/2013', 5000);

INSERT INTO Pub.tpcustomer VALUES(2000, 'cust 7', '01/06/2013', 5000);
```

```
ALTER TABLE Pub.tpcustomer

  PARTITION BY RANGE custid USING TABLE AREA "Customer_Table_Area"
  ( PARTITION partn_2k VALUES <= (2000))

      USING INDEX CustomerAlt5_idxc21, CustomerAlt5_idxc22;
```

At this stage table is partitioned with a single partition definition `partn_2k`, and all data is accessible.

The below statement add a new partition to have values less than or equal to 1000 and marks the partition as the split target:

```
ALTER TABLE Pub.tpcustomer
PREPARE FOR SPLIT PRO_INITIAL( PARTITION partn_1k VALUES <= (1000));
```

The below statement uses the PROUTIL utility to move data from the composite initial partition to target partition partn_1k.

```
$PROUTIL testdb -C partitionmanage split table Pub.tpcustomer composite initial useindex
 tpcustomer_idxc21
```

The above statement moves the rows satisfying condition custid<=1000 into a newly added partition. Now, the table contains one physical partition containing values <= 1000 and one composite partition contains values <= 2000 and > 1000.

The following statement splits without any partition definitions to move rest of data from the composite partition:

```
ALTER TABLE Pub.tpcustomer PREPARE FOR SPLIT PRO_INITIAL
```

The above statement marks partn_2k as the split target. Running the utility will move remaining records from the composite partition to partn_2k. In this case four records will be moved.

Migrating an unpartitioned table to a LIST-LIST partitioned table

```
CREATE TABLE Pub.sub_customer_list
 (
   Custid int,
   Custname varchar (50),
   Zipcode int
 )
   AREA "TP area";

CREATE INDEX sub_customer_list_list_idxlc1 ON Pub.sub_customer_list
 ( zipcode, custid, custname )
  AREA "TP idx area";

CREATE INDEX sub_customer_list_list_idxlc2 ON Pub.sub_customer_list
 (zipcode,custid)
  AREA "TP idx area";

CREATE INDEX sub_customer_list_list_idxg1 ON Pub.sub_customer_list (custid)
  AREA "TP idx area";

ALTER TABLE Pub.sub_customer_list
 PARTITION BY LIST zipcode
 SUBPARTITION BY LIST custid
 USING TABLE AREA "TP Area"
 (
  PARTITION sub_customer_lista1 VALUES IN (28019, 801),
  PARTITION sub_customer_lista2 VALUES IN (28019, 802),
  PARTITION sub_customer_listab VALUES IN (28019, 803),
  PARTITION sub_customer_listb1 VALUES IN (28028, 1804),
  PARTITION sub_customer_listb2 VALUES IN (28028, 1805),
  PARTITION sub_customer_listb3 VALUES IN (28028,1806),
  PARTITION sub_customer_listc1 VALUES IN (28039,2807),
  PARTITION sub_customer_listc2 VALUES IN (28039,2808),
  PARTITION sub_customer_listc3 VALUES IN (28039,2809))
  USING INDEX sub_customer_list_list_idxlc1, sub_customer_list_list_idxlc2;
```

The below statement marks all the partitions specified above as split targets:

```
ALTER TABLE Pub.sub_customer_list PREPARE FOR SPLIT PRO_INITIAL
```

Executing the `PROUTIL` utility, now, moves data from the composite initial partition to the target partition.

Migrating an unpartitioned table to a LIST-RANGE partitioned table

```
CREATE TABLE Pub.sub_customer_range
 (
   Custid int,
   Custname varchar (50),
   Zipcode int
 )
   AREA "TP area";

CREATE INDEX sub_customer_range_idxlc1 ON Pub.sub_customer_range
 ( zipcode, custid )
  AREA "TP Area";

CREATE INDEX sub_customer_range_idxgl ON Pub.sub_customer_range
 (custid)
  AREA "TP Area";

ALTER TABLE Pub.sub_customer_range
 PARTITION BY LIST zipcode
 SUBPARTITION BY RANGE custid
 USING TABLE AREA "TP Area"
 (
  PARTITION sub_customer_range1 VALUES <= (28019, 805),
  PARTITION sub_customer_range2 VALUES <= (28028, 1809),
  PARTITION sub_customer_range3 VALUES <= (28039, 2809))
  USING INDEX sub_customer_range_idxlc1 (ZipCode, custid);
```

In the above statement, the maximum value for the RANGE partition key column (`custid`) is specified for each leading `LIST` key column (`zipcode`)value.

The below statement further splits the `sub_customer_range1` partition:

```
ALTER TABLE Pub.sub_customer_range
PREPARE FOR SPLIT PRO_INITIAL
(
  PARTITION sub_customer_range1a VALUES <= (28019, 801),
  PARTITION sub_customer_range1b VALUES <= (28019, 802),
);
```

The below statement marks the `sub_customer_range1` partition as the split target:

```
ALTER TABLE Pub.sub_customer_range
PREPARE FOR SPLIT PRO_INITIAL
TARGET PARTITION sub_customer_range1;
```

ALTER TABLE syntax to add or drop partitions

Use the following syntax to add or drop partitions to or from a partitioned table:

Syntax

```
ALTER TABLE table name
  [ADD
  [(]
        PARTITION
    [partition name] VALUES { <= | IN } (column_values)
          [ RO_RW_ATTRIBUTE ][area_spec]
          [ PRO_DESCRIPTION desc],
     [PARTITION [partition name] VALUES { <= |IN } (column_values)
         [area_spec] [ PRO_DESCRIPTION desc],...]
  [)]
  ]
  [ DROP { PARTITION | PARTITIONS } partition name,
    [partition name, ...] ]
```

Parameters

column_values

Uses the following syntax:

```
column_value[ , column_value, ...]
```

column_value

Uses a constant as its value

RO_RW_ATTRIBUTE

Uses the following syntax and marks the partition as a read-only partition or a read-write partition:

```
READ_ONLY | READ_WRITE
```

If RO_RW_Attribute is not specified, then by default, the partition is marked as a read-write partition.

Notes

The following semantics apply when using ALTER TABLE to add and drop partitions:

- The table specified in the syntax must be a partitioned table
- In case of a table partitioned by LIST, the new partition must be specified with key values that are not already defined
- In case of a table partitioned by RANGE with data, the new RANGE partition can be added with a RANGE key value greater than the existing maximum key value of the table

- If the table is empty, new RANGE partitions can be added within the defined partition key values
- The partition to be dropped must be empty and must not be allocated
- The database truncate utility must be used to remove data from the partition and no allocation must be assigned to it before executing the ALTER TABLE DROP syntax
- The last partition can also be dropped
- To drop a partition, the table must contain the partition name specified
- To drop multiple partitions, the partition names must be specified in a comma-separated list and the table must contain those partitions
- Dropping a partition is not allowed when split, merge, or other database utilities are operating on a partition
- Dropping a partition which is referring to the initial composite partition is not allowed
- Dropping a partition is not allowed when it is marked as ready to be split
- Dropping a partition is not allowed when it is the source for a split operation
- A read-only partition cannot be deallocated, hence cannot be dropped.

Examples

Adding RANGE partitions

The following example illustrates adding RANGE partitions.

```
CREATE TABLE Pub.tpCustRange
  (
  Custid int,
  Custname varchar (50),
  Salary int
  )
 PARTITION BY RANGE custid
 USING TABLE AREA "Customer Table Area"
 (
 PARTITION partn20k VALUES <= (20000),
 PARTITION partn50k VALUES <= (50000),
 PARTITION partn80k VALUES <= (80000),
  );
```

The below statement adds a partition with values <= 30000.

```
ALTER TABLE Pub.tpCustRange
ADD PARTITION partn20k VALUES <= 30000;
```

Note: Adding new partition whose key values are less than the existing RANGE key values is allowed since the table does not contain any data.

The below statement adds two partitions whose values are <= 40000 and <= 90000, respectively:

```
ALTER TABLE Pub.tpCustRange
ADD
  (PARTITION partn40k VALUES <= 40000,
   PARTITION partn90k VALUES <= 90000,);
```

Adding partitions to a subpartitioned table

The below example shows how to add partitions to a subpartitioned table:

```
CREATE TABLE Pub.tpsub
  (
     a int,
  b int,
  c int,
  d int
 )
    PARTITION BY LIST a
  SUBPARTITION BY LIST b
  SUBPARTITION BY RANGE c
  USING TABLE AREA "Tenant 1 table Area"
    (
   PARTITION  sub_a values <= (1,1,100),
   PARTITION  sub_b values <= (2,2,200),
   PARTITION  sub_c values <= (3,3,100),
   PARTITION  sub_d values <= (3,3,200),
   PARTITION  sub_e values <= (3,3,300)
    );

ALTER TABLE Pub.tpsub
 ADD ( PARTITION sub2_e_1 VALUES <= (3,3,400),
       PARTITION sub2_e_2 VALUES <= (3,3,500))
```

Adding LIST partitions

The following example illustrates adding LIST partitions:

```
CREATE TABLE Pub.tpCustList
  (
  Custid int,
  Custname varchar (50),
  City varchar (50),
  join_date date,
  Salary int
  )
  PARTITION BY LIST city
  USING TABLE AREA "Customer Table Area"
    (
  PARTITION p1 VALUES IN ( 'Atlanta' ),
  PARTITION p2 VALUES IN ( 'Montgomery' ),
  PARTITION p3 VALUES IN ( 'Boston' )
    );
```

The below statement adds two new partitions:

```
ALTER TABLE Pub.tpCustList
 ADD
  (
  PARTITION p4 VALUES IN ( 'Jacksonville' ),
  PARTITION p5 VALUES IN ( 'Chicago')
  );
```

Adding read-only partitions

The following example illustrates adding a read-only partition:

```
ALTER TABLE Pub.TPCustomer ADD PARTITION cust04 VALUES IN (40, 41, 42) READ_ONLY;
```

Adding read-write partitions

The following example illustrates adding a read-write partition:

```
ALTER TABLE Pub.TPCustomer ADD PARTITION cust06 VALUES IN (50, 51, 52) READ_WRITE;
```

Dropping partitions

The following example illustrates dropping a single partition:

```
ALTER TABLE Pub.tpCustList
DROP PARTITIONS p4;
```

The following example illustrates dropping multiple partitions:

```
ALTER TABLE Pub.tpCustList
DROP PARTITIONS p4, p5;
```

Authorization

To alter a table, you must have DBA privileges, ownership of the table, or all the specified privileges on the table.

Related statements

ADD TABLE, DROP TABLE

Altering partitioned tables without partition schema definition

Syntax to alter a table to be a partitioned table without any partition schema definition

```
ALTER TABLE table_name SET PARTITION [ TABLE ]
   [ USING INDEX index_name, index_name, ...]
```

Syntax to create partition policies on a partitioned table without any partition schema definition

```
ALTER TABLE ...partition_policy
   [(partition_policy_detail, [partition_policy_detail], ...)]
```

Parameters

partition_policy

Uses the following syntax:

```
PARTITION BY {RANGE|LIST} column_name
  [ PRO_DESCRIPTION desc]
  [subpartition_definition,...]area_spec
```

partition_policy_detail

Uses the following syntax:

```
PARTITION
  [partition_name] VALUES { <= |IN }( column_values )
        [area_spec][ PRO_DESCRIPTION desc ]
```

Notes

- To alter a partitioned table without a partition schema definition, the table must be empty and its indexes and LOB columns must be in the type II area. All the indexes specified in the `USING INDEX` clause are marked as local and must have common prefix keys; The rest of the indexes are marked as global.
- Partition policy definitions can be also be added using the `ALTER TABLE ADD PARTITION` statement. If a table has only a row-based default index, then the default index is removed and a new default index with partitioned columns and row IDs is created.

Examples

Altering a table to be marked as a partitioned table without any partition schema definition

```
CREATE TABLE Order
 (
    orderid integer,
    Item varchar (50),
    Order_date date,
    Country varchar (50),
    State varchar (50),
    City varchar (50))
    AREA "area_type2";

CREATE INDEX order_idx1 ON Order (order_date)
    AREA "area_type2";

CREATE INDEX order_idx2 ON Order (Item)
    AREA "area_type2";

ALTER TABLE Order SET PARTITION
    USING INDEX order_idx1;
```

Executing the above statement marks the `Order` table as a partitioned table without any partition schema definition and the index `order_idx1` is marked as a local index.

Altering a partitioned table without any partition schema definition to be a partitioned table with partition policy definitions

The following example illustrates altering a partitioned table without any partition schema definition to a partitioned table with partition policy definitions using the ALTER TABLE ADD PARTITION statement:

```
ALTER TABLE Order PARTITION BY RANGE (order_date)
USING TABLE AREA "area_type2";
ALTER TABLE Order ADD
(
  PARTITION P2014 VALUES <= (2014),
  PARTITION P2015 VALUES <= (2015));
```

ALTER TENANT

The ALTER TENANT statement can be used to:

- Modify the definition of a tenant
- Change the NO SPACE parameter specified for one or more existing multi-tenant tables
- Add an existing tenant to an existing multi-tenant group that is defined for a multi-tenant table

The table name and group name specified in the FOR TABLE JOIN clause should not be repeated. You can add the tenant to only one group for a given table. However, you can add it to multiple groups on different tables.

The current tenant partition for the table specified in the FOR TABLE clause should be allocated NO SPACE. When the tenant for the specified table joins a group, its existing table, index, and LOB partitions for that table is removed. No new partition is created for the tenant-table combination.

Syntax

```
ALTER TENANT tenant_name
 [FOR TABLE [owner_name] table_name_1
    [{[USING TABLE AREA table_area_name]
   [ USING INDEX AREA index_area_name]
   [ USING LOB AREA lob_are_name]
      | JOIN GROUP group_name }]...;
```

Parameters

tenant_name

Specifies the name of the tenant you want to modify.

FOR TABLE

Indicates the default area, which can be overridden by a specific area for a table, or for its index, or for its LOB areas. You can specify an area of storage for data of tables and indexes.

TABLE AREA area_name

Is used to over-ride the default storage areas associated with the particular tenant. This area is used for database space allocation during the CREATE TABLE statement execution.The *TABLE AREA* specification is optional.

INDEX AREA index_area_name

Modifies the name of the default index storage area for a tenant.

If the storage area for an index is omitted in the syntax, then the default index area of the tenant is allocated to the specified index.

LOB AREA lob_area_name

Modifies the name of the default LOB storage area for a tenant.

If the storage area for a LOB is omitted in the syntax, then the default LOB area of the tenant will be allocated to the specified LOB.

table_name_1

Specifies the name of the table. The referenced table name must be a valid table in the multi-tenant database.

owner_name

Specifies the name of the schema that qualifies the table.

JOIN GROUP

Adds the tenant to the specified group.

group_name

Specifies the name of the group that you want the tenant to join.

Example

The following example modifies the table partition of the tenant to the allocated space and adds the tenant to a group:

```
 ALTER TENANT Mega_Media_Networks
     FOR TABLE pub.finance_accts USING TABLE DEFAULT AREA
     FOR TABLE pub.new_customers USING TABLE AREA "MMN Data area"
                                 USING INDEX AREA "MMN_Index_Area"

     FOR TABLE pub.farm_location
     FOR TABLE pub.archive_10yr JOIN GROUP Joint_Archives

;
```

Authorization

To alter a tenant, you must have DBA privileges.

Related Statement

DROP TENANT

ALTER USER

Changes the password for the specified user.

Syntax

```
ALTER USER { 'username' | 'username@domain_name' },
{FOR SQL ONLY | NOT FOR SQL ONLY};
ALTER USER { 'username' | 'username@domain_name' }
'old_password', 'new_password'
```

Parameters

username

Specifies the username. The username must be enclosed in single quotes.

domain_name

Specifies the domain name that the user is associated with. The username@domain_name must be enclosed in single quotes.

old_password

Specifies the current password for the user. The password must be enclosed in quotes.

new_password

Specifies a new password for the user. The password must be enclosed in quotes.

To set a new password without specifying the old password, use the following syntax:

```
ALTER USER 'username', 'new_password';
```

Notes

- Used in conjunction with CREATE USER and DROP USER, the ALTER USER statement provides a way to change a user password.
- The *old_password* specification must match the current password for *username*.
- The phrase FOR SQL ONLY is optional and if you use the phrase while issuing the ALTER USER statement, the user that is altered is used only by OE SQL. This user is no longer available to ABL and is referred to as a **SQL only user**. **SQL only users** do not affect ABL. If an OpenEdge Database contains only SQL only users, ABL treats the database as if there are no users in it. To revert this attribute, use the NOT FOR SQL

ONLY phrase and this causes the user to be a multi-purpose user. Only users with DBA priviliges can execute the **ALTER USER** statement with the FOR SQL ONLY or the NOT FOR SQL ONLY attribute.

Examples

ALTER USER to change the account password

In this example, the ALTER USER statement Jasper changes the Jasper account password from normandy to brittany:

```
ALTER USER 'Jasper', 'normandy', 'brittany' ;
```

ALTER USER to change the account password and make the user SQL ONLY

In this example, the ALTER USER statement Jasper changes the Jasper account password from normandy to brittany and also changes the user to a SQL ONLY user:

```
ALTER USER 'Jasper', 'normandy', 'brittany' FOR SQL ONLY;
```

ALTER USER to change the account password and change the user to Multi-purpose from SQL ONLY

In this example, the ALTER USER statement Jasper changes the Jasper account password from normandy to brittany and also changes the user to a Multi-purpose user from SQL ONLY user:

```
ALTER USER 'Jasper', 'normandy', 'brittany' NOT FOR SQL ONLY ;
```

ALTER USER to change the password for a user associated with a domain

The example displayed below shows how to change the password for a user associated with a domain. The following ALTER USER statement changes the password from normandy to brittany for the user Jasper associated with mtdomain domain:

```
ALTER USER 'Jasper@mtdomain', 'normandy', 'brittany' ;
```

Authorization

User specified in *username*.

Related statements

CREATE USER, DROP USER

AUDIT INSERT

Writes application audit events to an audit-enabled database.

Note: For more information about auditing, see *Learn about Security and Auditing*. For more information about enabling a database for auditing, see *Manage the OpenEdge Database*.

Syntax

```
AUDIT INSERT
(event_id,
  [event_context |NULL ]
  [event_detail |NULL ]);
```

Parameters

event_id

Positive integer value corresponding to an audit event record. The *event_id* must be a value greater than 32000.

event_context

Free-form character value that qualifies the *event_id*. May include non-ASCIIcharacters.

event_detail

Free-form character value that supplies detailed information about the audit event. May include non-ASCIIcharacters.

Notes

- Before inserting the specified application audit event into the database, the OpenEdge SQL engine determines the following:
  - The connected user has been granted the audit insert privilege.
  - The event id is valid and active.

  If both of these conditions are true, the engine writes the application audit event to the database. If one or both are not true, the engine does not write the event.
- `AUDIT INSERT` always returns a success status. This prevents users from determining whether or not they have privileges to log application audit events.

AUDIT SET

Allows grouping of audit data by the client on a per-connection basis.

Note: For more information about auditing, see *Learn about Security and Auditing*. For more information about enabling a database for auditing, see *Manage the OpenEdge Database*.

Syntax

```
AUDIT SET { EVENT_GROUP | APPLICATION_CONTEXT }
          {string | NULL }, {string | NULL };
```

Parameters

EVENT_GROUP

Indicates that subsequent audit records written by the database engine during the current connection will be marked as part of an event group.

APPLICATION_CONTEXT

Indicates that subsequent audit records written by the database engine during the current connection will be saved with application context information.

string

Free-form character value which is a unique string identifier for the group/application context event.

NULL

Clears an event group or application context string.

string

Free-form character value which provides additional application detail that describes the group/application context.

NULL

Clears an event group or application context string.

Note

AUDIT SET always returns a success status. This prevents users from determining whether or not they have audit privileges.

Examples

Setting an application context

In this example, an application context is set:

```
AUDIT SET APPLICATION_CONTEXT 'app.name.checking' '06/02/2005 Deposits';
```

Clearing an application context

In this example, the application context is cleared:

```
AUDIT SET APPLICATION_CONTEXT NULL NULL;
```

Authorization

Must have AUDIT_ADMIN, AUDIT_ARCHIVE, or AUDIT_INSERT privileges.

CALL

Invokes a stored procedure.

Note: SQL limits recursive invocation of a stored procedure to five levels.

Syntax

```
CALL proc_name ( [parameter][ , ...] );
```

Parameters

proc_name

The name of the procedure to invoke.

parameter

Literal or variable value to pass to the procedure.

Example

This example shows an excerpt from an ODBC application that calls a stored procedure (order_parts) using the ODBC syntax {call *procedure_name (param)*}:

```
SQLUINTEGER Part_num;
SQLINTEGER  Part_numInd = 0;
// Bind the parameter.
     SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT,
          SQL_C_SLONG, SQL_INTEGER, 0, 0, &Part_num, 0, Part_numInd);
// Place the department number in Part_num.
Part_num = 318;
// Execute the statement.
SQLExecDirect(hstmt, "{call order_parts(?)}", SQL_NTS);
```

Authorization

Must have DBA or EXECUTE privileges.

Related statements

CREATE PROCEDURE, DROP PROCEDURE

Column constraints

Specifies a constraint for a column that restricts the values that the column can store. INSERT, UPDATE, or DELETE statements that violate the constraint fail. The database returns a constraint violation error with an SQLCODE of -20116.

Column constraints are similar to table constraints, but their definitions are associated with a single column rather than the entire table.

Syntax

```
CONSTRAINT constraint_name
  NOT NULL [ PRIMARY KEY | UNIQUE ]
     | REFERENCES [owner_name.]table_name
        [ ( column_name ) ]
     | CHECK ( search_condition )
```

Parameters

CONSTRAINT *constraint_name*

Allows you to assign a name for the column constraint. This option facilitates making changes to the column definition. If you do not specify a *constraint_name*, the database assigns a name. These names can be long and unwieldy, and you must query system tables to retrieve the name.

NOT NULL

Restricts values in the column to values that are not null.

NOT NULL PRIMARY KEY

Defines the column as the primary key for the table. There can be at most one primary key for a table. A column with the NOT NULL PRIMARY KEY constraint should not contain null or duplicate values.

Other tables can name primary keys as foreign keys in their REFERENCES clauses. If they do, SQL restricts operations on the table containing the primary key in the following ways:

- DROP TABLE statements that delete the table fail
- DELETE and UPDATE statements that modify values in the column that match a foreign key's value also fail

NOT NULL UNIQUE

Defines the column as a unique key that cannot contain null or duplicate values. Columns with NOT NULL UNIQUE constraints defined for them are also called candidate keys.

Other tables can name unique keys in their REFERENCES clauses. If they do, SQL restricts operations on the table containing the unique key.

REFERENCES *table_name* [(*column_name*)]

Defines the column as a foreign key and specifies a matching primary or unique key in another table. The REFERENCES clause names the matching primary or unique key.

A foreign key and its matching primary or unique key specify a referential constraint. A value stored in the foreign key must either be null or be equal to some value in the matching unique or primary key.

You can omit the *column_name* argument if the table specified in the REFERENCES clause has a primary key and you want the primary key to be the matching key for the constraint.

CHECK (*search_condition*)

Specifies a column-level check constraint. SQL restricts the form of the search condition. The search condition must not:

- Refer to any column other than the one with which it is defined
- Contain aggregate functions, subqueries, or parameter references

Example

Creating a primary key

The following example shows the creation of a primary key column on the supplier table:

```
CREATE TABLE supplier (
     supp_no     INTEGER CONSTRAINT supp_key_con NOT NULL PRIMARY KEY,
     name        CHAR (30),
     status      SMALLINT,
     city        CHAR (20)
) ;
```

Creating a constraint to define the column as a unique key

The following example creates a NOT NULL UNIQUE constraint to define the column ss_no as a unique key for the employee table:

```
CREATE TABLE employee (
     empno        INTEGER NOT NULL PRIMARY KEY,
     ss_no        INTEGER NOT NULL UNIQUE,
     ename        CHAR (19),
     sal          NUMERIC (10, 2),
     deptno       INTEGER NOT NULL
) ;
```

Defining a foreign key that references the primary key

The following example defines order_item.orditem_order_no as a foreign key that references the primary key orders.order_no:

```
CREATE TABLE orders (
     order_no INTEGER NOT NULL PRIMARY KEY,
     order_date DATE
) ;
 CREATE TABLE order_item (
     orditem_order_no INTEGER REFERENCES orders ( order_no ),
     orditem_quantity INTEGER
) ;
```

The second CREATE TABLE statement in the previous example could have omitted the column name order_no in the REFERENCES clause, since it refers to the primary key of table orders.

Creating a check constraint

The following example creates a check constraint:

```
CREATE TABLE supplier (
     supp_no     INTEGER NOT NULL,
     name        CHAR (30),
     status      SMALLINT,
     city        CHAR (20) CHECK (supplier.city <> 'BadApple')
) ;
```

If a column is defined with a UNIQUE column constraints, no error results if more than one row has a NULL value for the column.

COMMIT

Commits a transaction explicitly after executing one or more SQL statements. Committing a transaction makes permanent any changes made by the SQL statements.

Syntax

```
COMMIT [ WORK ] ;
```

Rules to use the COMMIT statement

The following rules should be followed while using the COMMIT statement:

- It can be used in an embedded SQL application.
- It can be used in tools like SQL Explorer which expose the COMMIT operation as a SQL statement.
- It cannot be used in an ODBC or JDBC application because ODBC exposes the COMMIT operation as the ODBC API SQLEndTran() and JDBC exposes the COMMIT operation in the connection object as the commit() method.
- It cannot be used in an OE SQL stored procedure because a stored procedure runs on the OpenEdge SQL server while transactions can be controlled only from a SQLclient.

Notes

- The SQL statements executed prior to executing the `COMMIT` statement are executed as one atomic transaction that is recoverable and durable. The transaction is serializable if you specify this isolation level.
- On a system failure and/or the execution of the `ROLLBACK`, the transaction is rolled back to its initial state. Any changes made by the transaction are undone, restoring the database to its initial state. In the event of a system failure, the transaction will be rolled back during crash recovery when the database is restarted.
- A `COMMIT` operation makes any database modifications made by that transaction permanent.
- Once a `COMMIT` operation is executed, the database modifications cannot be rolled back.
- Once a `COMMIT` operation is executed, the transaction modifications are guaranteed durable regardless of any transient system failures.
- The atomicity applies only to the database modification and not to any direct I/O performed to devices such as the terminal, printer, and OS files by the application code.
- A `COMMIT` operation releases all locks implicitly or explicitly acquired by the transaction.

Related Statement

ROLLBACK

CONNECT AS CATALOG

Establishes a connection to an auxiliary read-only database.

Syntax

```
CONNECT 'database_path' AS CATALOG catalog_name;
```

Parameters

database_path

Full path to database directory and database name. This must be contained within quotes.

catalog_name

Catalog name to be used as an alias for the database in schema, table and column references. This must be in the form of an SQL identifier.

Notes

- Used to provide read-only access to multiple databases on a single client connection to an SQL server.
- Once connected, the catalog name for an auxiliary database may be used in SQL statements to qualify schema, table, and column access.
- The catalog name is visible, and usable, only in the client-server session in which it is defined.
- The catalog name of the primary database is the name of the primary database. The database name is the name by which the database is started (for example, by the `proserve` command) omitting all file system path information.

- Several client-server sessions may each connect to the same auxiliary database within an entire OpenEdge SQL Server process. Each such client-server session may use the same or different name when connecting to the same auxiliary database.
- If you are connected to a primary database that is unencrypted and it was started using the -t startup parameter, you will be unable to simultaneously connect to an encrypted auxiliary database. An attempt to do so will result in an error.

Example

In this example, the database named `customer` in directory `'/usr/databases'` is connected as a catalog named `mydb1`:

```
CONNECT 'usr/databases/customer' AS CATALOG mydb1;
```

Authorization

Any user allowed to execute this statement. However, authorization for access to the auxiliary database is governed by the same rules that govern access to the primary database. That is, the username and password of the current user must be authorized using access control information in the auxiliary database.

SQL Compliance

Progress Software Corporation specific extension.

Related statements

DISCONNECT CATALOG, SET CATALOG

CREATE DOMAIN

Defines a security domain to authenticate users for an existing tenant or a super-tenant. The domain name must be unique within the set of domain names that are defined for a database. A tenant can be associated with multiple domains.

Syntax

```
CREATE DOMAIN domain_name
  FOR { TENANT tenant_name | SUPER_TENANT super-tenant_name }
      [ PRO_DESCRIPTION value ];
```

Parameters

domain_name

Specifies the name of the security domain. You must specify the domain name even if it is the same as the tenant name.

TENANT tenant_name

Specifies the name of an existing tenant. For a DEFAULT tenant, you must specify `default` as the tenant name.

`SUPER_TENANT super-tenant_name`

Specifies the name of the tenant defined as a super-tenant in the database.

`PRO_DESCRIPTION` *value*

Allows you to enter an ABL description. The *value* attribute is an arbitrary character string.

Example

The following example illustrates how to create a tenant and a domain for the tenant.

To define a tenant called `Mega_Media_Networks`:

```
CREATE TENANT  Mega_Media_Networks TABLE AREA  "MMM Data area"
                                   INDEX AREA  "MM index area"
                                   LOB   AREA  "MMM pix area" ;
```

To define a domain for the tenant `Mega_Media_Networks`:

```
CREATE DOMAIN  dom_MMNet FOR TENANT Mega_Media_Networks;
```

To specify users who can securely connect to the domain `dom_MMNet`, you must create users and associate them with the domain `dom_MMNet`.

Authorization

To create a domain, you must have Database Administrator (`DBA`) privileges.

Related Statement

DROP DOMAIN

CREATE GROUP

Creates a group for the specified table in a multi-tenant database. A multi-tenant group allows multiple tenants to map to the same table partition for a multi-tenant table and therefore share the same data for that table. All data within the table partition is shared by the multiple tenants that are part of the group.

You can create multiple groups for the same table. The areas specified in the command prompt must be pre-existing as valid Type II areas in the database.

Syntax

```
CREATE GROUP group_name
  FOR TABLE table_name TABLE AREA area_name_1
    [ INDEX AREA area_name_2 ]
    [ LOB AREA area_name_3 ]
    [ PRO_DESCRIPTION value ]
```

Parameters

group_name

Specifies the name of the group for the multi-tenant table. The group name must be unique across the database.

table_name

Specifies the table name. The name of a referenced table must be a valid multi-tenant table. The table name can be qualified by an owner (schema) name.

TABLE AREA *area_name*

Is used for database space allocation for the table, during the CREATE GROUP statement execution. The *TABLE AREA* must be specified.

INDEX AREA *index_area_name*

Is used for database space allocation for the index, during the CREATE GROUP statement execution. The *INDEX AREA* specification is optional. The name of the storage area must be specified within double quotes.

If the storage area for an index is omitted in the syntax, the *TABLE AREA* acts as its default area.

LOB AREA*lob_area_name*

Is used for database space allocation for LOB, during the CREATE GROUP statement execution.The *LOB AREA* specification is optional. The name of the storage area must be specified within double quotes.

If the storage area for a LOB is omitted in the syntax, the *TABLE AREA* acts as its default area.

PRO_DESCRIPTION *value*

Allows you to enter the ABL description. The *value* attribute is an arbitrary character string.

FOR TABLE

Specifies the name of the table for which the group is defined.

Example

The following example shows how to create a group for a table and allocate partition space for the table, index, and LOB:

```
CREATE GROUP carz_second FOR TABLE pub.customer
       [ TABLE AREA "CUSTTABAREA" ]
       [ INDEX AREA "CUSTIDAREA" ]
       [ LOB AREA "CUSTTLOBAREA" ]
       [ PRO_DESCRIPTION "Second Group for CARZ" ]
```

Authorization

To create a group, you must have DBA privileges.

Related Statement

SHOW GROUP

CREATE INDEX

Creates an index on the specified table using the specified columns of the table. An index improves the performance of SQL operations whose predicates are based on the indexed column. However, an index slows performance of INSERT, DELETE, and UPDATE operations.

A DBA must use the FOR TENANT phrase to over-ride the default index storage areas and allocate new database storage areas for the index. A multi-tenant index is generated for a multi-tenant table.

Syntax

```
CREATE [ UNIQUE ] INDEX index_name
 ON table_name ({column_name[ ASC | DESC ]}[, ...])
    [ AREA area_name]
    [ ENCRYPT WITH cipher]
    [BUFFER_POOL{PRIMARY|ALTERNATE}]
            [ PRO_DESCRIPTION value | PRO_ACTIVE {'N'|'n'} ];

[ FOR TENANT {tenant_name_1|DEFAULT}
    USING INDEX AREA area_name | USING NO SPACE ]...;
```

Parameters

UNIQUE

Does not allow the table to contain rows with duplicate column values for the set of columns specified for that index.

index_name

Must be unique for the given table.

table_name

The name of the table on which the index is built.

column_name **[, ...]**

The columns on which search and retrieval is ordered. These columns are called the index key. When more than one column is specified in the CREATE INDEX statement, a concatenated index is created.

ASC **|** DESC

Allows the index to be ordered as either ascending (ASC) or descending (DESC) on each column of the concatenated index. The default is ASC.

AREA *area_name*

The name of the storage area where the index and its entries are stored. The name of the index area must be specified within double quotes. If the specified area does not exist, the database returns an error.

If you do not specify the index area, then the indexes are created in the default index area. You can configure the default index area using following command.

```
proutil db-name -C setobjectdefaultarea
“Customer index Area”  index
```

ENCRYPT WITH *cipher*

Allows the index to be encrypted by designating an appropriate cipher.

BUFFER_POOL **{**PRIMARY **|** ALTERNATE**}**

Allows the index to be assigned to a primary or alternate buffer pool.

PRO_DESCRIPTION *value*

Allows you to enter an ABL description. *value* is an arbitrary character string.

PRO_ACTIVE **{**'N' **|** 'n'**}**

Indicates the creation of an inactive index. Inactive indexes can be created for an online database.

FOR TENANT *tenant_name*

Specifies the name of a tenant. The FOR TENANT phrase can be repeated for as many existing tenants as required.

The FOR phrase cannot be used, if it is not a multi-tenant table.

USING INDEX AREA area_ name

Is used to over-ride the default storage areas associated with the particular tenant. This area is used for database space allocation during the CREATE INDEX statement execution. The *INDEX AREA* specification is optional.

If this phrase is not specified, the index will be allocated in the default index area for the particular tenant.

Note: The area name is ignored for a multi-tenant index.

USING NO SPACE

Indicates that no space is allocated for index data storage for the table. This phrase is valid only if the tenant partition for the table is allocated NO SPACE.

Note: For a particular tenant, the table and the Index partitions must have the same allocation state.

Notes

- The first index you create on a table should be the fundamental key of the table. This index cannot be dropped except by dropping the table.
- An index slows down the performance of INSERT, DELETE, and UPDATE operations.
- Use PROUTIL to activate indexes.
- Use CREATE INDEX without the PRO_ACTIVE { 'N' | 'n' } attribute to create an inactive index. Active indexes can be created for an online database if the following conditions are met:
  - Run CREATE INDEX immediately after creating a table.
  - The index is created on the table.
  - Both the CREATE TABLE and CREATE INDEX are performed within the same transaction (no commit is performed after CREATE TABLE is run).

Examples

Creating a unique index

The following example illustrates how to create a unique index on a table:

```
CREATE UNIQUE INDEX custindex ON customer ( cust_name );
```

Creating an inactive word index

The following example shows how to create an inactive word index with the specified description field:

```
CREATE PRO_WORD INDEX CommentsWordIdx ON pub.customer ( cust_name )
     PRO_DESCRIPTION 'Word index on customer name field'
     PRO_ACTIVE 'n';
```

Creating a multi-tenant index

The following example shows how to define a multi-tenant index:

```
CREATE PRO_WORD INDEX CommentsWordIdx ON mtcustomer ( cust_name )
    MULTI_TENANT
       FOR TENANT DEFAULT USING INDEX AREA "Misc_Index_Area";
```

Creating indexes for partitioned tables

You can use the CREATE INDEX statement to create partitioned indexes. The following types of indexes are available for partitioned tables:

- Local index
- Global index

These indexes are selected based on the keys on which the index is defined.

Local Index

A local index must include the partition key column as the leading prefix of its index keys. If there are *n* partition key columns, then the first *n* index key components must be the partition key columns. This is also known as a partition aligned index.

Use the following syntax to specify area definitions for each index partition:

Syntax

```
CREATE [ UNIQUE ] INDEX index_name ON table_name...
    [ FOR PARTITION partition_name USING INDEX AREA area_name]...
```

Global Index

A global index is a single database object, which contains index entries for all rows, in all partitions, in a partitioned table. For a partitioned table, all indexes that are not partition aligned are global indexes. A partition aligned index can be a local index or a global index. By default, all partition aligned indexes are local indexes. To create global partition aligned indexes, use the GLOBAL reserved keyword, as shown below:

Syntax

```
CREATE [ GLOBAL ][ UNIQUE ] INDEX index_name
  ON table_name...
       [ AREA area_name]...
```

Notes

- The following semantics apply when using the CREATE INDEX statement for partitioned tables:
  - The partition specified in the FOR PARTITION clause must be an existing partition in the table
  - Partitions can be specified in any order

- The specified area must be present in the table and must be in the type II area
- The table specified in the statements must be a partitioned table

- The following semantics are applicable to local indexes:
  - The existing AREA clause cannot be used for local indexes.
  - The FOR PARTITION clause is only valid for a partitioned table and a local index.
  - Only index areas can be specified in the FOR PARTITION clause in the above syntax.
  - The specified area must be a type II area.
  - The first real local index created for a partitioned table without a partition schema definition is marked as the default index.
  - The initial composite partition of a partitioned table without a partition schema definition is created with NO SPACE for local indexes.

- The following semantics are applicable to global indexes:
  - If an AREA clause is not specified, the index is created in the default index area of a table partition. A global index can use the area defined in an AREA clause for data in all partitions.
  - An index created using an unpartitioned column as the prefix of index keys is treated as a global index.
  - If an AREA clause is not specified, the index is created in the default index area of a table partition.
  - A global index can use an existing AREA clause.
  - The FOR PARTITION clause cannot be used for creating global indexes.
  - The GLOBAL clause must be used only for partitioned tables. If it is used to create indexes on any other tables (like shared tables or multi-tenant tables), the SQL engine returns an error.
  - The GLOBAL clause can also be used to create a global index for a partitioned table without partition schema definition. The AREA clause must be specified for a global index of a partitioned table without partition schema definition and this area must belong to the type II area.

Examples

Creating local indexes

The following example illustrates creating local indexes for the partitioned table Pub.tpcustomer with partitions:

```
CREATE INDEX custid_localidx on Pub.tpcustomer (custid,salary)
FOR PARTITION P1 USING INDEX AREA "custidxp1"
FOR PARTITION P2 USING INDEX AREA "custidxp2"
FOR PARTITION P3 USING INDEX AREA "custidxp3";
```

Creating local indexes on a table with partition-specific index areas

The following example creates a local index on the Pub.tpcustomer table with partition specific index areas for partitions partn1 and partn9.

```
CREATE INDEX cust_localidx on Pub.tpcustomer ( custid )
FOR PARTITION partn1 USING INDEX AREA "idx1 part1 area"
FOR PARTITION partn9 USING INDEX AREA "idx1 part2 area";
```

Creating global indexes

The following example illustrates creating a global index using the AREA clause:

```
CREATE INDEX custname_globallidx on Pub.tpcustomer (custname)
 AREA "custidx";
```

Creating global indexes on a table with a partitioning key column

The following example illustrates creating a global index, where `custid` is the partitioning key for the table `Pub.tpcustomer`:

```
CREATE GLOBAL INDEX custname_globalidx2 on Pub.tpcustomer (custid)
 AREA "custidx";
```

Creating a unique global index

The following example illustrates creating a unique global index, where `orderid` is the partitioning key for the table `Pub.tporderid`:

```
CREATE GLOBAL UNIQUE INDEX custname_globalidx3 on Pub.tporderid (orderid)
 AREA "custidx";
```

Authorization

To create a index, you must have DBA privileges or the `INDEX` privilege on the table.

Related statements

ALTER TABLE, CREATE TABLE, DROP INDEX

CREATE PROCEDURE

Creates a stored procedure. Stored procedures contain a Java code snippet that is processed into a Java class definition and stored in the database in text and compiled form. SQL applications invoke stored procedures through the SQL `CALL` statement or the procedure-calling mechanisms of ODBC and JDBC.

Syntax

```
CREATE PROCEDURE [owner_name.]procname
( [parameter_decl[ , ...]] )
  [ RESULT ( column_name data_type[ , ...] ) ]
  [ IMPORT
      java_import_clause]
   BEGIN
      java_snippet
   END
```

Parameters

parameter_decl

This is the syntax for *parameter_decl*:

```
{ IN | OUT | INOUT }parameter_namedata_type
```

owner_name

Specifies the owner of the procedure. If the name is different from the user name of the user executing the statement, then the user must have DBA privileges.

procname

Names the stored procedure. DROP PROCEDURE statements specify the procedure name defined here. SQL also uses *procname* in the name of the Java class that it creates from the Java snippet.

IN | OUT | INOUT

Specifies whether the parameter declaration is input, output, or both.

Calling applications pass values for input parameters in the CALL statement or CALL escape sequence.

Stored procedures assign values to output parameters as part of their processing.

INOUT parameters have both a value passed in and receive a new value during procedure processing.

parameter_name data_type

Names a parameter and associates an SQL data type with it. The data type must be one supported by OpenEdge.

RESULT (*column_namedata_type*[, ...])

Specifies columns in the result set the procedure returns. If the CREATE PROCEDURE statement includes this clause, the Java snippet must explicitly insert rows into the result set using the Java class SQLResultSet.

Note that the *column_name* argument is not used in the body of the stored procedure. Instead, methods of the Java classes refer to columns in the result set by ordinal number, not by name. The IMPORT keyword must be uppercase and on a separate line. The body is a sequence of Java statements between the BEGIN and END keywords. The Java statements become a method in a class that SQL creates and submits to the Java compiler. The BEGIN and END keywords must be uppercase and on separate lines.

Notes

SQL limits recursive invocation of a stored procedure to five levels.

See *Develop SQL for OpenEdge* for more information on using the CREATE statement and stored procedures.

Example

The following example illustrates the use of the CREATE PROCEDURE statement:

```
CREATE PROCEDURE get_sal ()
IMPORT
import java.math.*;
BEGIN
StringBuffer ename = new StringBuffer (20) ;
BigDecimal esal = new BigDecimal (2) ;
SQLCursor empcursor = new SQLCursor (
"SELECT name, sal FROM emp " ) ;
empcursor.open () ;
empcursor.fetch ();
while (empcursor.found ())
{ename = (StringBuffer) empcursor.getValue (1, CHAR);
esal = (BigDecimal) empcursor.getValue (2, NUMERIC);
// do something with the values here
}empcursor.close () ;
END
```

Authorization

Must have DBA privilege, RESOURCE privilege, or ownership of procedure.

Related statements

CALL, DROP PROCEDURE

CREATE SEQUENCE

A sequence is an object for creating an incremental number series. Sequences can generate sequential values within any integer range with either positive or negative increments. The database holds the sequence definition and keeps track of the next available value.

To create a multi-tenant sequence, use the MULTI_TENANT attribute after defining all the other sequence attributes in the CREATE SEQUENCE syntax.

Syntax

```
CREATE SEQUENCE [schema_name] sequence_name
    [INCREMENT BY value],
    [START WITH value],
    [MAXVALUE value | NOMAXVALUE],
    [MINVALUE value | NOMINVALUE],
    [CYCLE | NOCYCLE]
    [MULTI_TENANT];
```

Parameters

schema_name

Specifies the schema to contain the sequence. If *schema_name* is not specified, the sequence generator creates the sequence in the current schema. OpenEdge supports only the PUBLIC (PUB) schema.

sequence_name

Specifies the name of the sequence to be created.

INCREMENT BY

Specifies the interval between sequence numbers. The value can be a positive or negative integer (INTEGER data type for 32-bit sequences, BIGINT datatype for 64-bit sequences), but it cannot be 0. The value range for a 32-bit sequence is -2,147,483,648 to 2,147,483,647. The value range for a 64-bit sequence is -9223372036854775808 to 9223372036854775807. The value is positive for an ascending sequence and negative for a descending sequence. The default value is 1.

START WITH

Specifies the first number in the sequence generated. The value must be greater than or equal to the MINVALUE for an ascending sequence. The value must be less than or equal to the MAXVALUE for a descending sequence. The default value for an ascending sequence is MINVALUE and for a descending sequence it is MAXVALUE.

MAXVALUE

Specifies the maximum value to be generated for the sequence. The default value is -1 for a 32-bit and a 64-bit descending sequence. The default value is 2,147,483,647 for a 32-bit ascending sequence and 9223372036854775807 for a a 64-bit ascending sequence.

NOMAXVALUE

Specifies -1 as the MAXVALUE for a 32-bit descending sequence and 2,147,483,647 as the MAXVALUE for a 32-bit ascending sequence. Specifies -1 as the MAXVALUE for a 64-bit descending sequence and 9223372036854775807 as the MAXVALUE for a 64-bit ascending sequence.

MINVALUE

Specifies the minimum value that the sequence can generate. The default value is 0 for an ascending sequence. The default value is -2,147,483,648 for a 32-bit descending sequence and -9223372036854775808 for a 64-bit sequence.

NOMINVALUE

Specifies 0 as the MINVALUE for an ascending sequence. The MINVALUE is -2,147,483,648 for a 32-bit descending sequence and -9223372036854775808 for a 64-bit sequence.

CYCLE

Indicates that the sequence will continue to generate a value after reaching the MAXVALUE (for an ascending sequence)or MINVALUE (for a descending sequence).

NOCYCLE

Indicates that the sequence cannot generate a value after reaching the MAXVALUE (for an ascending sequence) or MINVALUE (for a descending sequence). The SQL sequence generator uses NOCYCLE as the default, if CYCLE is not specified.

MULTI_TENANT

Specifies the sequence defined as a multi-tenant sequence.

Examples

Generating unique customer numbers

The following example shows how a sequence generates unique customer numbers when a new customer is inserted into the pub.customer table:

```
CREATE SEQUENCE pub.customer_sequence
     START WITH 100,
     INCREMENT BY 1,
     NOCYCLE;
```

Defining a multi-tenant sequence

The following example shows how to define a multi-tenant sequence:

```
CREATE SEQUENCE pub.mtcustomer_sequence
     START WITH 100,
     INCREMENT BY 1,
     NOCYCLE
     MULTI_TENANT
     ;
```

Authorization

To create a sequence, you must have DBA privileges.

Related Statements

ALTER SEQUENCE, DROP SEQUENCE

CREATE SUPER-TENANT

Defines a tenant for a multi-tenant database. It also allocates database resources in the existing multi-tenant tables.

Defines a super-tenant for a multi-tenant database. A super-tenant is a user who can access and maintain data for any tenant defined in the multi-tenant database. There can be multiple super-tenant users created within a single database.

The multi-tenant database resources are not allocated automatically when you create a super-tenant. Privileges of a regular tenant, such as creating, selecting, inserting, updating, or deleting database objects are also applicable to the super-tenant.

The syntax for creating a super-tenant is given below:

Syntax

```
CREATE SUPER_TENANT super_tenant_name [ PRO_DESCRIPTION value ];
```

Parameters

super-tenant_name

Specifies the name of the super-tenant. You must ensure that the name is unique within the set of tenants in a database.

PRO_DESCRIPTION *value*

Allows you to enter an ABL description. The *value* attribute is an arbitrary character string.

Example

The following example shows how to create a SUPER_TENANT*tenantadmin*:

```
CREATE SUPER_TENANT tenantadmin;
```

Authorization

To create a super-tenant, you must have DBA privileges.

Related Statement

CREATE TENANT

CREATE SYNONYM

Creates a synonym for the specified table, view, or synonym. A synonym is an alias that SQL statements can use instead of the name specified when the table, view, or synonym was created.

Syntax

```
CREATE [ PUBLIC ] SYNONYM synonym  FOR
[owner_name.]{table_name | view_name | synonym} ;
```

Parameters

`PUBLIC`

Specifies that the synonym is public: all users can refer to the name without qualifying it. By default, the synonym is private: other users must qualify the synonym by preceding it with the user name of the user who created it.

Users must have the DBA privilege to create public synonyms.

`SYNONYM` *`synonym`*

Name for the synonym.

`FOR` `[`*`owner_name.`*`]{`*`table_name`*` | `*`view_name`*` | `*`synonym`*`}`

Table, view, or synonym for which SQL creates the new synonym.

Example

The following example demonstrates the use of the `CREATE SYNONYM` statement:

```
CREATE SYNONYM customer FOR smith.customer ;
CREATE PUBLIC SYNONYM public_suppliers FOR smith.suppliers ;
```

Authorization

Must have `DBA` privilege or `RESOURCE` privilege.

Related Statement

DROP SYNONYM

CREATE TABLE

Creates a table definition. A table definition consists of a set of named column definitions for data values that will be stored in rows of the table. SQL provides two forms of the `CREATE TABLE` statement.

The CREATE TABLE syntax can be used to:

- Explicitly specify the definition of a column.
- Implicitly define the columns using the columns in a query expression with the AS*query_expression* clause.
- Define a new table as a multi-tenant table, and allocate storage area of a database to the tenants.
- Specify partition key details along with area definitions.
- Create a partitioned table with no partition schema defined

Syntax



Parameters

column_definition

Defines a column. The syntax for column_definition is given below:

```
column_namedata_type
  [ COLLATE case_insensitive |case_sensitive ]
  [ DEFAULT {literal| NULL | SYSDATE | SYSTIME | SYSTIMESTAMP }]
  [column_constraint[column_constraint , ...]]
  [progress_column_attribute_keyword value ], ...]
```

column_name data_type

Names a column and associates a data type to it. The name of the column must be different from other column names already defined in the table. The *data_type* must be supported by OpenEdge.

When a table contains more than one column, a comma separator should be used after each *column_definition*, except for the final *column_definition*.

COLLATE

Indicates the case sensitivity of the column. The default value for the COLLATE element is *case_sensitive*.

case_insensitive

Indicates that the column will be case-insensitive. The word *case_insensitive* itself cannot be used as a valid input. The value for the *case_insensitive* clause can only be _I, I, or the default database collation with the suffix _I (For example: COLLATE_I, COLLATE I,or COLLATE BASIC_I).

case_sensitive

Indicates that the column will be case-sensitive. The word *case_sensitive* itself cannot be used as a valid input. The value for the *case_sensitive* clause can only be _S, S, or the default database collation with the suffix _S (For example: COLLATE_S, COLLATE S,or COLLATE BASIC_S).

DEFAULT

Specifies an explicit default value for a column. The column takes on the default value if an INSERT statement does not include a value for the column. If a column definition omits the DEFAULT clause, NULL value gets assigned.

The DEFAULT clause accepts the arguments shown in the table below:

Argument	Description
literal	An integer, numeric, or a string constant.
NULL	A null value.

Argument	Description
SYSDATE	Displays the current date. Valid only for columns defined with the DATE data type. SYSDATE is equivalent to the Progress default keyword TODAY. The argument represents a DATE value.
SYSTIME	Displays the current time. The argument represents a TIME value.
SYSTIMESTAMP	Displays the current date and time. The argument represents a TIMESTAMP value.

column_constraint

Specifies a constraint that is applied when you insert or update a value in the associated column.

progress_column_attribute_keyword value

ABL column attribute keyword and value.

The syntax used to define a LOB column is shown below:

```
{ LVARCHAR | CLOB | LVARBINARY | BLOB }
  [ ( length ) ]
  [ AREA area_name ]
  [ ENCRYPT WITH cipher ]
  [ BUFFER_POOL { PRIMARY | ALTERNATE } ]
```

table_constraint

Specifies a constraint that is applied when you insert or update a row in the table.

AREA *area_name*

Specifies the name of the storage area where a particular object is to be stored. The name of the storage area must be specified within double quotes. If the specified area does not exist, the database returns an error.

If you do not specify an area, then the default area as defined in the database schema. The default database areas are stored in first three elements of array field _db. _db-misc1 and you can configure them using database utility proutil.

You can set the default area for TABLE, INDEX, and LOB using the proutil command:

```
proutil db-name -C setobjectdefaultarea area-name [ object-type | all ]
```

In order to query the default area, you can use the following command:

```
proutil db-name -C dispobjectdefaultarea [ object-type | all ]
```

ENCRYPT WITH *cipher*

Allows the table to be encrypted by designating an appropriate cipher.

BUFFER_POOL {PRIMARY | ALTERNATE}

Allows the table to be assigned to a primary or alternate buffer pool.

progress_table_attribute_keyword value

ABL table attribute keyword and value.

AS *query_expression*

Specifies a query expression to be used for the data type and data values for the columns of the table. The type and length of the columns of the query_ expression result become the type and length of the respective columns in the table you created. The rows in the resultant set of the query_ expression are inserted into the table after its creation. The column names are optional in this form of the CREATE TABLE statement. If it is omitted, the names of the columns of the table are taken from the column names of the query_ expression.

MULTI_TENANT

Defines the table as a multi-tenant table. The MULTI_TENANT phrase can occur only after the *column_definition* and the *progress_table_attribute_keyword*, and before the AS query_expression part of the CREATE TABLE syntax.

FOR TENANT *tenant_name*

Specifies the name of a tenant. The FOR TENANT phrase can be repeated for as many existing tenants as required.

TABLE AREA *area_name*

Is used to over-ride the default storage areas associated with the particular tenant. This area is used for database space allocation during the CREATE TABLE statement execution. Specifying the *TABLE AREA* is optional.

Specifies the name of the default storage area allocated for the tenant.

INDEX AREA index *_area_name*

Is used to over-ride the default storage areas associated with the particular tenant. This area is used for database space allocation during the CREATE TABLE statement execution. Specifying the *INDEX AREA* is optional.

If the storage area for an index is omitted in the syntax, then any index will be allocated in the table area.

To define *INDEX AREA*, you must define the *TABLE AREA*.

LOB AREA lob *_area_name*

Is used to over-ride the default storage areas associated with the particular tenant. This area is used for database space allocation during the CREATE TABLE statement execution. Specifying the *LOB AREA* is optional.

If the storage area for a LOB is omitted in the syntax, then any LOB will be allocated in the table area.

To define the *LOB AREA*, you must define the *TABLE AREA*.

PRO_DESCRIPTION *value*

Allows you to enter ABL description. *value* is an arbitrary character string.

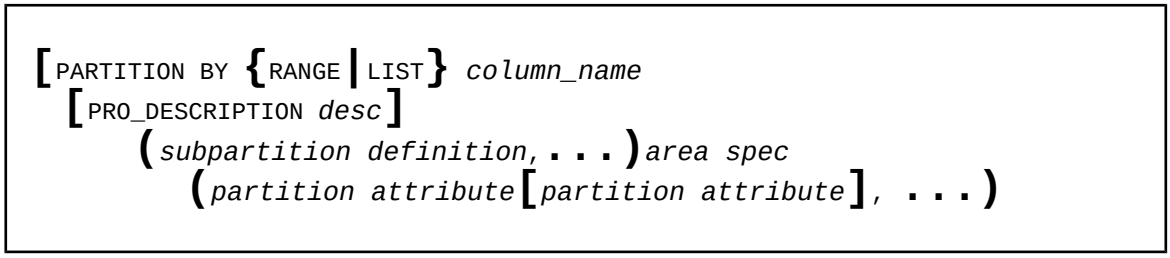
USING NO SPACE

Indicates that no space is allocated for index data storage in the table. This phrase is valid only if the tenant partition for the table is allocated NO SPACE.

Note: For a particular tenant, the table and the Index partitions must have the same allocation state.

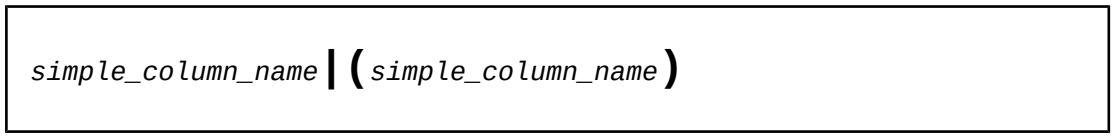
partition definition

Uses the following syntax:



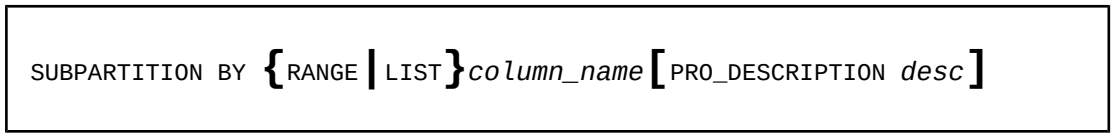
column_name

Uses the following syntax:



subpartition definition

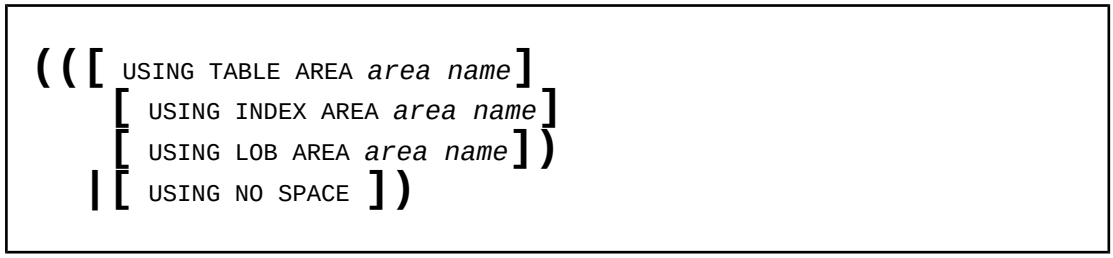
Uses the following syntax:



area spec

Specifies the default areas for all partitions defined, if not overridden by specific partition definitions.

The syntax for *area spec* is:



partition attribute

Uses the following syntax:

```
PARTITION
  [partition_name] VALUES {<= | IN}(column_values)
       [ RO_RW_ATTRIBUTE ][area spec][PRO_DESCRIPTION desc]
```

partition_name

Specifies the SQL identifier.

column_values

Uses the following syntax:

```
column_value[ , column_value, ...]
```

column_value

Uses a constant as its value

RO_RW_ATTRIBUTE

Uses the following syntax and marks the partition as a read-only partition or a read-write partition:

```
READ_ONLY | READ_WRITE
```

If RO_RW_Attribute is not specified, then by default, the partition is marked as a read-write partition.

Note: A NO SPACE partition cannot be marked as a read-only partition.

Notes

- Table columns defined in OpenEdge SQL have default format values identical to those created by the Data Dictionary. Thus, columns created by SQL will have the same default format as columns created by ABL tools.
- The following semantics apply when using CREATE TABLE for partitioned tables:
  - The partitioned database objects must be defined only in Type II storage areas.
  - A database can be enabled for both multi-tenancy and table partitioning, but a table cannot be both multi-tenant and partitioned.
  - Expressions and scalar functions cannot be used as partition keys and key columns.
  - The PARTITION BY clause does not allow column lists.

- Partition names are optional, unique across a database, and have the lexical properties of a table name; if not specified, system generated partition names are used.
- The maximum number of partitions per table is 32,767.
- A partition can be defined without any allocated data storage.
- <= indicates that, for the partition being defined, the values of the partition key columns in data rows are less than or equal to the value specified for that partition in the CREATE TABLE statement. The partition keys values in data rows are greater than the values specified for the previous partition definition.
- Partition keys must be literal constants and must be enclosed in parentheses for LIST partitions. For RANGE partitions, the parentheses are optional.
- All index supported data types can be used as partition keys.
- The existing AREA clause in the CREATE TABLE statement cannot be used if a PARTITION BY clause is used.
- Descriptions can be added using the CREATE TABLE statement in the following two places:
- After the PARTITION BY clause (as a description for all partitions)
- At the end of each partition definition (as a description for that particular partition, overriding any description provided at upper levels)
- The area definition at the lower level always takes priority among the area definitions provided at multiples levels.
- All INSERT statements with partition key column values that match the values specified in the partition definition belong to the corresponding partition. If none of the partition definitions match the key column value in an INSERT statement, then that INSERT statement returns an error. For a LIST partition, the column values specified by an INSERT statement must match the partition key value. For a RANGE partition, the column values must be less than or equal to the highest partition key value defined. This ensures that the new table row maps to one of the defined partitions.
- The partition policy name has the table name suffixed with the table ID. If the table name is more than 20 characters long, only the first 20 characters from the table name are used.
- A partition key column cannot be dropped or renamed.
- A column cannot be removed from the partition key definition of the partitioned table.
- A column's datatype cannot be changed (applicable to any column; not just a partition key column).

- The following semantics apply when using CREATE TABLE for subpartitioning:
  - Up to 15 levels of subpartitioning are allowed.
  - Partition key columns must not be repeated.
  - Parentheses around partition key columns are optional.
  - For a subpartition, all the leading subpartition levels must define LIST partitions. The last subpartition level can be either a RANGE partition or a LIST partition.
  - If a partition definition has a RANGE column then the <= clause must be used to specify partition key values, otherwise use the IN clause.
  - For RANGE partitions, all partitions having the same LIST key values must be specified sequentially with the RANGE key values in ascending order.
  - The partition definition must contain values for all partition keys.
  - Each *column_value* corresponds to the column name of the partition or subpartition, in the same order in which the column names are defined.

- The following semantics apply when using CREATE TABLE to create a partitioned table with no partition schema defined:
  - The partition key columns and partition definitions must not be specified.
  - A default ROWID index is created for a partitioned table with no partition schema defined.
  - The AREA clause is optional; if specified, it must be a type II area.
  - The initial composite partition of the table and LOB columns are created with NO SPACE.

Examples

CREATE TABLE statement

The following example shows that the user who issues the CREATE TABLE statement must have REFERENCES privilege on the itemno column of the table john.item in CREATE TABLE supplier_item:

```
CREATE TABLE supplier_item
 (
    supp_no     INTEGER NOT NULL PRIMARY KEY,
    item_no     INTEGER NOT NULL REFERENCES john.item ( itemno ),
    qty         INTEGER
 ) ;
```

CREATE TABLE statement

The table is created in the current owner schema.

The following CREATE TABLE statement explicitly specifies a table owner, gus:

```
CREATE TABLE gus.account
 (
    account  integer,
    balance  numeric (12,2),
    info     char (84)) ;
```

CREATE TABLE statement to load a table with a subset of the data in another table

The following example shows the AS*query_expression* form of CREATE TABLE to create and load a table with a subset of the data in the customer table:

```
CREATE TABLE dealer (name, street, city, state)
     AS
          SELECT name, street, city, state
          FROM customer
          WHERE state IN ('CA','NY', 'TX') ;
```

CREATE TABLE statement to include a column constraint

The following example includes a NOT NULL column constraint and DEFAULT clause for definition of a column:

```
CREATE TABLE emp
(
     empno  integer NOT NULL,
     deptno  integer DEFAULT 10,
     join_date date DEFAULT NULL
 ) ;
```

CREATE TABLE statement to create a table with two columns

The following example shows how to create a table with two columns. Both of them have ABL descriptions and specified column labels:

```
CREATE TABLE emp
(
     empno INTEGER NOT NULL UNIQUE
       PRO_DESCRIPTION 'A unique number for each employee'
       PRO_COL_LABEL 'Employee No.'
     deptno INTEGER DEFAULT 21 NOT NULL
       PRO_DESCRIPTION 'The department number of the employee'
       PRO_COL_LABEL 'Dept. No.'
 )PRO_HIDDEN 'Y' PRO_DESCRIPTION 'All Employees';
```

A hidden table is created and it has a specified description.

CREATE TABLE statement to create a multi-tenant table that overrides areas of the selected tenant table partition

The following example shows how to create a multi-tenant table that overrides areas of the selected tenant table partition:

```
CREATE TABLE pub.acct_payable (acct_num    integer, debit_date   date)
   MULTI_TENANT
    FOR TENANT Consolidated_Freightways
               USING TABLE AREA "CF Financial area"
               USING INDEX AREA "CF Fin idx area"
    FOR TENANT "Mega Media Networks"
               USING TABLE AREA "MMN Data area"
               USING INDEX AREA "MMN index area"
               USING LOB   AREA "MMN pix area" ;
     FOR TENANT DEFAULT USING NO SPACE ;
```

CREATE TABLE statement to create a multi-tenant table that uses the default area for all the table partition of a tenant

The following example shows how to a create multi-tenant table which uses the default area for all the table partition of a tenant. In this case, the DEFAULT tenant is allocated no space in the database storage area:

```
CREATE TABLE pub.mtcustomer (cust_num    integer, hire_date   date)
 MULTI_TENANT;
```

CREATE TABLE statement to create a multi-tenant table that uses the default area for all the area tenants except the default tenant

The following example shows how to create a multi-tenant table which uses the default area for all areas tenants except the DEFAULT tenant:

```
CREATE TABLE pub.mtcustomer (cust_num    integer, hire_date   date)
   MULTI_TENANT
     FOR TENANT DEFAULT USING TABLE AREA "Scratch_Data_Area"
                      USING INDEX AREA "Misc_Index_Area";
```

CREATE TABLE statement to create partitioned tables

The following example illustrates partitioning a table based on customer ID. It specifies the default table area for each partition. Values less than or equal to 1000 will be a part of PARTITION p1, values ranging from 1001 to 2000 will be a part of PARTITION p2, and values ranging from 2001 to 3000 will be a part of PARTITION p3. Using an INSERT statement to insert values greater than or equal to 3000 returns an error.

```
CREATE TABLE Pub.tpcustomer
 (
  custid int,
  custname VARCHAR (50),
  join_date date,
  salary int
 )
  PARTITION BY RANGE custid USING TABLE AREA "area_pt"
   (
     PARTITION p1 VALUES <= (1000) USING TABLE AREA "area_p1",
     PARTITION p2 VALUES <= (2000) USING TABLE AREA "area_p2",
     PARTITION p3 VALUES <= (3000) USING TABLE AREA "area_plast"
   );
```

CREATE TABLE statement to partition a table by LIST

The following example illustrates partitioning a table by LIST.

```
CREATE TABLE Pub.tpcustomer
 (
    custid int,
    custname VARCHAR (50),
    city VARCHAR (10),
    salary int
 )
     PARTITION BY LIST CITY
      USING TABLE AREA "custtab_area"
      USING INDEX AREA "custidx_area"
     (
        PARTITION p1 VALUES IN ( 'Atlanta' ),
        PARTITION p2 VALUES IN ( 'Montgomery' ),
        PARTITION p3 VALUES IN ( 'Boston' ));
```

In the above example, partitions p1, p2, and p3 contain customers whose cities are Atlanta, Montgomery, and Boston, respectively.

Creating constraints on a partitioned table

The following examples illustrate creating constraints on a partitioned table.

Note: If a constraint includes partition key column as the leading prefix of its constraint columns, then a local index is created for the constraint, otherwise, a global index is created. For more information about local and global indexes, see the section.

```
CREATE TABLE Pub.tpcustomer
  (
  "cust_num" int,
  "SalesRep" VARCHAR (160),
  "tp_date" date primary key,
  "r_value" int
 )
 PARTITION BY RANGE "tp_date"
 USING TABLE AREA "Tenant 1_table_area"
 PARTITION "custtab_area"
 USING INDEX AREA "custidx_area"
  (
  PARTITION tpcustomer_p1 VALUES <= ( '12/31/2011' ),
  PARTITION tpcustomer_p2 VALUES <= ( '12/31/2012' ),
  PARTITION tpcustomer_p3 VALUES <= ( '12/31/2013' )
    );
```

A local index is created for the constraints in the above example.

```
CREATE TABLE Pub.tpcustomer
  (
  "cust_num" int,
   Name VARCHAR (60) UNIQUE,
   Address VARCHAR (160),
   Zip VARCHAR (160),
   "SalesRep" VARCHAR (160),
   "tp_date" date,
   "r_value" int
 )
 PARTITION BY RANGE "tp_date"
 USING TABLE AREA "Tenant 1_table_area"
  (
  PARTITION tpcustomer_p1 VALUES <= ( '12/31/2011' ),
  PARTITION tpcustomer_p2 VALUES <= ( '12/31/2012' ),
  PARTITION tpcustomer_p3 VALUES <= ( '12/31/2013' )
    );
```

A global index is created for the constraints in the above example.

Subpartitioning a table

The following example illustrates subpartitioning and creates LIST-LIST-LIST partitions on the tporder_list table:

```
CREATE TABLE tporder_list
 (orderid integer,
  Item varchar(50),
  Order_date date,
  Country varchar(50),
  State varchar(50),
  City varchar(50))
  PARTITION BY LIST (Country)
  SUBPARTITION BY LIST(State)
  SUBPARTITION BY LIST(City)
  USING TABLE AREA "order list table area"
     (  PARTITION USA_MA_BT VALUES IN (‘USA','MA','Boston'),
    PARTITION USA_NY_NY VALUES IN (‘USA','NY','New York'),
      USING INDEX AREA "Secunderabad index area"
    PARTITION USA_MA_BD VALUES IN (‘USA,'MA','Bedford'))
```

Creating a table with LOB partitions

The following example shows how to create a table with LOB partitions:

```
CREATE TABLE Pub.tpcustomer
 (
  F1 INTEGER,
  F2 VARCHAR
 )
 PARTITION BY RANGE F1
 USING TABLE AREA "Tenant 1 table Area"
 (
  PARTITION Pub.tpcustomer_p1 VALUES <= (1000)
       USING LOB AREA "Partn misc lob Area",
  PARTITION Pub.tpcustomer_p2 VALUES <= (2000),
  PARTITION Pub.tpcustomer_p3 VALUES <= (3000);
```

LOB partitions are created in areas as per the areas specified in the partition definition. In the above example, for LOB column F2, for partition Tab1_P1, the LOB partition area is Partn misc lob Area; For the rest of the partitions, the LOB partition area is Tenant 1 table Area.

The AREA phrase is not allowed for LOB columns while creating partitioned tables and the LOB columns cannot be partition key columns.

Creating a table with read-only partitions

The following example illustrates creating a table with read-only partitions:

```
CREATE TABLE Pub.tpcustomer
 (
   a int,
   b int,
   c int,
   d int
 )
 PARTITION BY LIST a
 SUBPARTITION BY LIST b
 SUBPARTITION BY LIST c
 USING TABLE AREA "Tenant 1_table_area"
  (
    PARTITION sub1_a VALUES IN ( 10, 11, 12) READ_ONLY,
    PARTITION sub1_b VALUES IN ( 20, 21, 22),
    PARTITION sub1_c VALUES IN ( 30, 31, 32) READ_ONLY
  );
```

Creating a table without any partition schema definition

```
CREATE PARTITION TABLE order
(
   orderid integer,
   Item varchar (50),
   Order_date date,
   Country varchar (50),
   State varchar (50),
   City varchar (50))
 AREA "area_type2";
```

Authorization

To create a table, you must have DBA privileges, and the RESOURCE privilege.

Related Statement

DROP TABLE

CREATE TENANT

The CREATE TENANT statement can be used to:

- Define a tenant for a multi-tenant database
- Allocate database resources in the existing multi-tenant tables
- Add a new tenant to an existing multi-tenant group that is defined for a multi-tenant table

The table name and group name specified in the FOR TABLE clause of the CREATE TENANT statement cannot be repeated in another instance of the FOR TABLE clause of the same statement. You can add the tenant to only one group for a given table. However, you can add it to multiple groups on different tables. No new partition is created for the table specified in the FOR TABLE clause.

Syntax

```
CREATE TENANT tenant_name TABLE AREA area_name
  [ INDEX AREA area_name_2 ]
  [ LOB AREA area_name_3 ]
  [ PRO_DESCRIPTION value ]
  [ FOR TABLE [owner_name] table_name_1
   {[ USING TABLE AREA area_name ]
    [ USING INDEX AREA index_area_name ]
    [ USING LOB AREA lob_area_name ] | USING NO SPACE | JOIN GROUP group_name
}]...
  [ FOR TABLE [owner_name] table_name_2[ USING NO SPACE ]];
```

Parameters

tenant_name

Specifies the name of the tenant for the multi-tenant table.

TABLE AREA *area_name*

Is used to over-ride the default storage areas associated with the particular tenant. This area is used for database space allocation during the CREATE TENANT statement execution.The TABLE AREA specification is optional. The name of the storage area must be specified within double quotes.

INDEX AREA *index_area_name*

Is used to over-ride the default storage areas associated with the particular tenant. This area is used for database space allocation during the CREATE TENANT statement execution. The *INDEX AREA* specification is optional. The name of the storage area must be specified within double quotes.

If the storage area for an index is omitted in the syntax, then any index will be allocated in the table area.

LOB AREA lob_ *area_name*

Is used to over-ride the default storage areas associated with the particular tenant. This area is used for database space allocation during the CREATE TENANT statement execution.The *LOB AREA* specification is optional. The name of the storage area must be specified within double quotes.

If the storage area for a LOB is omitted in the syntax, then any LOB will be allocated in the table area.

PRO_DESCRIPTION *value*

Allows you to enter the ABL description. The *value* attribute is an arbitrary character string.

FOR TABLE

Indicates the default area, which can be overridden by a specific area for a particular table, or for its index, or for its LOB areas. An area can be specified for the storage of data of table and index.

USING NO SPACE

Indicates that no space is allocated for index data storage for the table. This phrase is valid only if the tenant partition for the table is allocated NO SPACE.

Note: For a particular tenant, the table and the Index partitions must have the same allocation state.

table_name

Specifies the table name. The name of a referenced table must be valid for a multi-tenant database.

owner_name

Specifies the name of the schema that qualifies the table.

JOIN GROUP

Adds the tenant to the specified group.

group_name

Specifies the name of the group that you want the tenant to join.

AREA *area_name*

Specifies the name of the storage area where data of the table is to be stored. The name of the storage area must be specified within double quotes.

If the specified area does not exist, the database returns an error. If you do not specify an area, the default area is used.

Examples

CREATE TENANT statement

The following example shows how to create a tenant by overriding an area for the selected table partition and how to add the tenant to a group:

```
CREATE TENANT  Consolidated_Freightways TABLE AREA "CF Data area"
                                        INDEX AREA "CF index area"
  FOR TABLE pub.customer USING TABLE AREA "CF cust area"
                       USING INDEX AREA "CF cust idx area"
  FOR TABLE pub.acct_payable USING TABLE AREA "CF Financial area"
                           USING INDEX AREA "CF Fin idx area"
  FOR TABLE pub.farm_location USING NO SPACE
  FOR TABLE pub.archive_10yr  JOIN GROUP Joint_Archives;
```

CREATE TENANT statement

The following two examples use the default area for the partition of all the tenant tables:

```
CREATE TENANT  Mega_Media_Networks TABLE AREA  "MMM Data area"
                                   INDEX AREA  "MMM index area"
                                   LOB   AREA  "MMM pix area" ;
```

```
CREATE TENANT  Mega_Media_Networks TABLE AREA  "MMM Data area"
                                   INDEX AREA  "MMM index area"
                                   LOB   AREA  "MMM pix area"
  FOR TABLE pub.archive_10yr  USING NO SPACE;
```

Authorization

To create a tenant, you must have DBA privileges.

Related Statement

DROP TENANT

CREATE TRIGGER

Creates a trigger for the specified table. A trigger is a special type of automatically executed stored procedure that helps ensure referential integrity for a database.

Triggers contain Java source code that can use SQL Java classes to carry out database operations. Triggers are automatically activated when an INSERT, UPDATE, or DELETE statement changes the trigger's target table. The Java source code details what actions the trigger takes when it is activated.

Syntax

```
CREATE TRIGGER [owner_name.]trigname
 { BEFORE | AFTER }
 { INSERT | DELETE | UPDATE [ OF column_name[ , ...]]}
ON table_name
[ REFERENCING { OLDROW [ ,NEWROW ]| NEWROW [ ,OLDROW ]}]
 [ FOR EACH { ROW | STATEMENT }]
 [ IMPORT
    java_import_clause]
  BEGIN
    java_snippet
  END
```

Parameters

owner_name

Specifies the owner of the trigger. If the name is different from the user name of the user executing the statement, then the user must have DBA privileges.

trigname

Names the trigger. DROP TRIGGER statements specify the trigger name defined here. SQL also uses *trigname* in the name of the Java class that it creates from the Java snippet.

BEFORE | AFTER

Denotes the trigger action time. The trigger action time specifies whether the triggered action, implemented by *java_snippet*, executes BEFORE or AFTER the invoking INSERT, UPDATE, or DELETE statement.

INSERT | DELETE | UPDATE [OF *column_name* [, ...]]

Denotes the trigger event. The trigger event is the statement that activates the trigger.

If UPDATE is the triggering statement, this clause can include an optional column list. Only updates to any of the specified columns will activate the trigger. If UPDATE is the triggering statement and does not include the optional column list, then any UPDATE on the table will activate the trigger.

ON *table_name*

Identifies the name of the table where the trigger is defined. A triggering statement that specifies *table_name* causes the trigger to execute. *table_name* cannot be the name of a view.

REFERENCING OLDROW [, NEWROW] | NEWROW [, OLDROW]

Provides a mechanism for SQL to pass row values as input parameters to the stored procedure implemented by java_snippet. The code in java_snippet uses the getValue method of the NEWROW and OLDROW objects to retrieve values of columns in rows affected by the trigger event and store them in procedure variables. This clause is allowed only if the trigger specifies the FOR EACH ROW clause.

The meaning of the OLDROW and NEWROW arguments of the REFERENCING clause depends on whether the trigger event is INSERT, UPDATE, or DELETE. For example:

- INSERT...REFERENCING NEWROW means the triggered action can access values of columns of each row inserted. SQL passes the column values specified by the INSERT statement.
- INSERT...REFERENCING OLDROW is meaningless, since there are no existing values for a row being inserted. INSERT...REFERENCING OLDROW generates a syntax error.
- UPDATE...REFERENCING OLDROW means the triggered action can access the values of columns, before they are changed, of each row updated. SQL passes the column values of the row as it exists in the database before the update operation.
- DELETE...REFERENCING OLDROW means the triggered action can access values of columns of each row deleted. SQL passes the column values of the row as it exists in the database before the delete operation.
- DELETE...REFERENCING NEWROW is meaningless, since there are no new existing values to pass for a row being deleted. DELETE...REFERENCING OLDROW generates a syntax error.

- UPDATE is the only triggering statement that allows both NEWROW and OLDROW in the REFERENCING clause.
- UPDATE...REFERENCING NEWROW means the triggered action can access the values of columns, after they are changed, of each row updated. SQL passes the column values specified by the UPDATE statement.
- The trigger action time (BEFORE or AFTER) does not affect the meaning of the REFERENCING clause. For instance, BEFORE UPDATE...REFERENCING NEWROW still means the values of columns after they are updated will be available to the triggered action.
- The REFERENCING clause generates an error if the trigger does not include the FOR EACH ROW clause.

FOR EACH { ROW | STATEMENT }

Controls the execution frequency of the triggered action implemented by *java_snippet*.

FOR EACH ROW means the triggered action executes once for each row being updated by the triggering statement. CREATE TRIGGER must include the FOR EACH ROW clause if it also includes a REFERENCING clause.

FOR EACH STATEMENT means the triggered action executes only once for the whole triggering statement. FOR EACH STATEMENT is the default.

IMPORT *java_import_clause*

Specifies standard Java classes to import. The IMPORT keyword must be uppercase and on a separate line.

BEGIN

java_snippet

END

Denotes the body of the trigger or the triggered action. The body contains the Java source code that implements the actions to be completed when a triggering statement specifies the target table. The Java statements become a method in a class that SQL creates and submits to the Java compiler.

The BEGIN and END keywords must be uppercase and on separate lines.

Notes

- Triggers can take action on their own table so that they invoke themselves. SQL limits such recursion to five levels.
- You can you have multiple triggers on the same table. Multiple UPDATE triggers on the same table must specify different columns. SQL executes all triggers applicable to a given combination of table, trigger event, and action time.
- The actions carried out by a trigger can fire another trigger. When this happens, the other trigger's actions execute before the rest of the first trigger finishes executing.
- If a constraint and trigger are both invoked by a particular SQL statement, SQL checks constraints first, so any data modification that violates a constraint does not also fire a trigger.
- To modify an existing trigger, you must delete it and issue another CREATE TRIGGER statement. You can query the systrigger system table for information about the trigger before you delete it.

- The code in java_snippet uses the `getValue` method of the `NEWROW` and `OLDROW` objects. The `getValue` method is valid on `OLDROW` before or after an update or delete and `NEWROW` before or after an update or insert; the `setValue` method is only valid on `NEWROW` before an insert or update.

Examples

CREATE TRIGGER statement

The following code segment illustrates how to use the `CREATE TRIGGER` statement:

```
CREATE TRIGGER TRG_TEST04 BEFORE INSERT ON tst_trg_01
REFERENCING NEWROW
FOR EACH ROW
IMPORT
   import java.sql.*;
BEGIN
   //Inserting Into tst_trg_03
   Integer new_value=newInteger(0);
   new_value=(Integer)NEWROW.getValue (1,INTEGER);
   SQLIStatement insert_tst3=new SQLIStatement ("INSERT INTO tst_trg_03 values (?)");
   insert_tst3.setParam (1,new_value);
   insert_tst3.execute();
END
```

CREATE TRIGGER statement

The following code segment illustrates how to set values for a new row in the `CREATE TRIGGER` statement:

```
CREATE TRIGGER trg1403
BEFORE INSERT ON tbl1401
REFERENCING NEWROW
FOR EACH ROW
IMPORT
import java.sql.* ;
BEGIN
INTEGER n2 = new INTEGER(12345);
NEWROW.setValue(2, n2);
END
```

For more information on creating and using triggers, see *Develop SQL for OpenEdge*.

Authorization

Must have the `DBA` privilege or `RESOURCE` privilege.

Related statement

DROP TRIGGER

CREATE USER

Creates the specified user.

Syntax

```
CREATE USER {'username' | 'username@domain_name'}, 'password' FOR SQL ONLY ;
```

Parameters

username

Specifies the username. The username must be enclosed in single quotes.

domain_name

Specifies the domain name that the user is associated with. The username@domain_name must be enclosed in single quotes.

password

Specifies the password for the user. The password must be enclosed in single quotes.

Notes

- You are strongly advised to NOT create a user named PUB. A user named PUB is inherently the owner of all tables created in the ABL and all schema tables, since these are all in the PUB schema. As the owner, a user PUB has full access to those tables, including the ability to read and write data, and the ability to drop the application table. Therefore, the existence of a user PUB creates a very serious security risk for the database. For these reasons, please do NOT create a user named PUB.
- Used in conjunction with BEGIN-END DECLARE SECTION and DROP USER statement, the CREATE USER statement provides a way to manage user records through SQL.
- The user name and password must be enclosed in quotes.
- The phrase FOR SQL ONLY is optional and if you use this phrase while issuing the CREATE USER statement, the user that is created is used only by OE SQL. This user is not available to ABL and is referred to as a **SQL only user**. **SQL only users** do not affect ABL. If an OpenEdge Database contains only SQL only users, ABL treats the database as if there are no users in it.
- Before issuing the CREATE USER statement, there are no users defined in the user table and any user can log into the database.
- After issuing the CREATE USER statement, only users defined in the user table can log into the database.
- If you are creating a user in a multi-tenant table, the created user should be associated with the tenant identified by the domain name defined in the syntax.
- The username@domainame can be equivalently defined in OpenEdge ABL. For more information on defining a user name in ABL, see *ABL Reference*.

Examples

CREATE USER statement

In this example an account with DBA privileges creates the '*username*' 'Jasper' with password 'spaniel':

```
CREATE USER 'Jasper', 'spaniel' ;
```

CREATE USER statement with FOR SQL ONLY

In the following example, CREATE USER creates the user Jasper to connect to mtdomain domain with the password spaniel and changes the user to a SQL only user:

```
CREATE USER 'Jasper@mtdomain', 'spaniel' FOR SQL ONLY ;
```

The user Jasper should be associated with the tenant identified by the domain mtdomain.

Authorization

Must have DBA privileges.

Related statements

BEGIN-END DECLARE SECTION, DROP USER

CREATE VIEW

Creates a view with the specified name on existing tables or views.

Syntax

```
CREATE VIEW [owner_name.]view_name
 [ ( column_name, column_name,... ) ]
 AS [ ( ]query_expression[ ) ]
  [ WITH CHECK OPTION ] ;
```

Parameters

owner_name

Owner of the created view.

(*column_name, column_name,...*)

Specifies column names for the view. These names provide an alias for the columns selected by the query specification. If the column names are not specified, then the view is created with the same column names as the tables or views on which it is based.

WITH CHECK OPTION

Checks that the updated or inserted row satisfies the view definition. The row must be selectable using the view. The WITH CHECK OPTION clause is only allowed on an updatable view.

To determine the authorization (privileges) of a view and to enable effective employment of the view, you must execute two GRANT statements along with the CREATE VIEW statement. The GRANT statements give access to the view and enables the view to access the underlying database tables.

The syntax for these statements is given below:

```
CREATE VIEW ...
GRANT {privilege[, privilege] , ... | ALL [ PRIVILEGES]}
ON schema_name view_name TO
  { username [, username ] , ... | PUBLIC }
GRANT {privilege[, privilege] , ... | ALL [ PRIVILEGES]}
ON table_name TO schema_name;
```

privilege

Uses the following syntax:

```
{ SELECT | INSERT | DELETE | INDEX
   | UPDATE [( column , column , ... )
  [| REFERENCES [( column , column , ... )]}
```

Notes

- A view is deletable if deleting rows from that view is allowed. For a view to be deletable, the view definition must satisfy the following conditions:
  - The first FROM clause contains only one table reference or one view reference.
  - There are no aggregate functions, DISTINCT clause, GROUP BY clause, or HAVING clause in the view definition.
  - If the first FROM clause contains a view reference, then the view referred to is deletable.
- A view is updatable if updating rows from that view is allowed. For a view to be updatable, the view has to satisfy the following conditions:
  - The view is deletable (it satisfies all the previously specified conditions for deletability).
  - All the select expressions in the first SELECT clause of the view definition are simple column references.
  - If the first FROM clause contains a view reference, then the view referred to is updatable.
- A view is insertable if inserting rows into that view is allowed. For a view to be insertable, the view has to satisfy the following conditions:
  - The view is updatable (it satisfies all the previously specified conditions for update ability).
  - If the first FROM clause contains a table reference, then all NOT NULL columns of the table are selected in the first SELECT clause of the view definition.
  - If the first FROM clause contains a view reference, then the view referred to is insertable.

Examples

CREATE VIEW statement

The following examples illustrate CREATE VIEW statements defined by query expressions:

```
CREATE VIEW ne_customers AS
          SELECT name, address, city, state
          FROM customer
          WHERE state IN ( 'NH', 'MA', 'ME', 'RI', 'CT', 'VT' )
          WITH CHECK OPTION ;
CREATE VIEW OrderCount (custnum, numorders) AS
          SELECT CustNum, COUNT(*)
          FROM Order
          GROUP BY CustNum;
```

Granting privileges on a view

The below statement creates a view `mtInvoiceView1` for the table `pub.mtinvoice`:

```
CREATE VIEW mtInvoiceView1 AS
SELECT CustNum,InvoiceDate,Amount,TotalPaid,Adjustment,OrderNum,ShipCharge from
pub.mtinvoice;
```

The below statement grants only `SELECT` privileges on the view:

```
GRANT SELECT PRIVILEGES ON Schema1.mtInvoiceView1 TO PUBLIC;
```

The below statement grants only `UPDATE` privileges on the view:

```
GRANT UPDATE PRIVILEGES ON Schema1.mtInvoiceView1 TO PUBLIC;
```

The below statement grants the schema that owns the view the authorization to select data from the table that the view accesses. Other privileges can also be granted. All table accessed by the view must be granted to the schema that owns the view.

```
GRANT ALL PRIVILEGES ON pub.mtInvoice TO Schema1;
```

Authorization

Must have `DBA` privilege, `RESOURCE` privilege, or `SELECT` privilege.

Related statements

DROP VIEW

DELETE

Deletes zero, one, or more rows from the specified table that satisfy the search condition specified in the `WHERE` clause. If the optional `WHERE` clause is not specified, then the `DELETE` statement deletes all rows of the specified table.

When deleting row(s) of a multi-tenant table, a regular tenant can only delete rows in its partition, but a super-tenant or a DBA can delete rows in all the tenant partitions and group partitions. A super-tenant or a DBA may refine which tenants' rows must be affected by using the `tenantid_tbl()` or the `tenantName_tbl()` function in the `WHERE` clause `search_condition`.

While deleting a row of a partitioned table, if the row belongs to a read-only partition, the `DELETE` statement returns an error.

Syntax

```
DELETE FROM [owner_name]{table_name|view_name}[ WHERE search_condition];
```

Notes

If the table has primary or candidate keys and there are references from other tables to the rows to be deleted, the statement is rejected.

Examples

DELETE statement

The following example illustrates the `DELETE` statement:

```
DELETE FROM Customer WHERE Name = 'Surf and Sport';
```

Deleting a row from a multi-tenant table

The following example deletes the row from a multi-tenant table using `custnum` and `tenantid_tbl()` in the `WHERE` clause.

```
DELETE FROM mtcustomer
WHERE custnum = 1428 AND tenantName_tbl (pub.mtcustomer) = 'SNCSoftware';
```

Authorization

To delete a row from the table, you must have `DBA` privileges, ownership of the table, or `DELETE` permission of the table.

A user mapped to a super-tenant can view data for all the tenants only if the user has the required permissions on the tables referenced in the statement.

Related statements

WHERE clause

DISCONNECT CATALOG

Removes a connection to an auxiliary read-only database.

Syntax

```
DISCONNECT CATALOG catalog_name;
```

Parameter

catalog_name

Catalog name to be used as an alias for the database in schema, table and column references. This must be in the form of an SQL identifier.

Notes

Used to remove auxiliary connections established by executing the `CONNECT AS CATALOG` statement.

Example

In this example, the auxiliary database connection identified by the catalog named `mydb1` is removed:

```
DISCONNECT CATALOG mydb1;
```

Authorization

Any user is allowed to execute this statement.

SQL Compliance

Progress Software Corporation specific extension

Related statements

CONNECT AS CATALOG, SET CATALOG

DROP DOMAIN

Using the drop domain, you can delete a security domain.

Syntax

```
DROP DOMAIN domain_name;
```

Parameters

domain_name

Specifies the name of the security domain.

Note: To drop a domain, you must first drop all the users associated with the domain.

Note

Each database contains tenants, users, and domains. A DBA must complete the steps specified below to drop a tenant from the database:

1. DROP USER for every user in the domain. For more information on dropping a user.
2. DROP DOMAIN for every domain defined for the tenant.
3. DROP TENANT.

Example

The syntax shown below shows how to drop a domain.

```
DROP DOMAIN jasper;
```

Authorization

To drop a domain, you must have DBA privileges.

Related Statement

CREATE DOMAIN

DROP GROUP

Deletes a group from a multi-tenant database and deallocates the database resources in the group partition of the multi-tenant table for which the group is defined.

Syntax

```
DROP GROUP group_name;
```

Parameter

group_name

Specifies the name of the group that you want to delete from the multi-tenant database.

Example

The following example illustrates the DROP GROUP statement:

```
DROP GROUP carz_group;
```

Note

The specified group must be an existing valid group in the database. All the tenants must be disassociated from the group before dropping it.

Authorization

You must have DBA privilege to drop a group.

Related statements

CREATE GROUP, SHOW GROUP

DROP INDEX

Deletes an index on the specified table.

If a multi-tenant index is dropped, then the DROP INDEX statement deletes the index for every tenant defined in the table.

You can also use the DROP INDEX statement to drop an index on a partitioned table and delete all the index partitions corresponding to the index being dropped. However, the index created for a constraint on the table cannot be dropped.

Syntax

```
DROP INDEX [index_owner_name]index_name
    ON [table_owner_name]table_name  ;
```

Parameters

index_owner_name

Specifies the name of the index owner. If index_owner_name is specified and is different from the name of the user executing the statement, then the user must have DBA privileges.

index_name

Verifies the *index_name* corresponding to the table.

Examples

DROP INDEX statement

The following example illustrates the DROP INDEX statement:

```
DROP INDEX custindex ON customer;
```

You cannot drop the first index created on a table, except by dropping the table.

Dropping a local index

The following example drops the local index custnum including all its partitions on the table Pub.tpcustomer:

```
DROP INDEX custnum ON Pub.tpcustomer;
```

Dropping a global index

The following example drops the global index custname, which has only one physical partition and that physical partition is dropped:

```
DROP INDEX custname ON Pub.tpcustomer;
```

Authorization

Must have DBA privilege or ownership of the index.

Related statement

CREATE INDEX

DROP PROCEDURE

Deletes a stored procedure.

You can use the DROP PROCEDURE statement as an online and offline operation. For databases enabled for OpenEdge Replication the DROP PROCEDURE statement is an offline operation. For databases without OpenEdge Replication the DROP PROCEDURE statement is an online operation.

Syntax

```
DROP PROCEDURE [owner_name.]procedure_name ;
```

Parameters

owner_name

Specifies the owner of the procedure.

procedure_name

Name of the stored procedure to delete.

Example

The following example illustrates the DROP PROCEDURE statement:

```
DROP PROCEDURE new_sal ;
```

Authorization

Must have DBA privilege or owner of a stored procedure.

Related statement

CALL, CREATE PROCEDURE

DROP SEQUENCE

The DROP SEQUENCE statement removes a sequence from a schema. The sequence can be in a user's schema or another schema may be specified. You must have DBA privileges to remove a sequence in a schema other than your own.

If a multi-tenant sequence is dropped, then the DROP SEQUENCE statement drops the sequence instance for every tenant defined in the multi-tenant table.

Syntax

```
DROP SEQUENCE [schema_name ]sequence_name;
```

Parameters

schema_name

Specifies the schema name that contains the sequence. If this is not specified, OpenEdge SQL drops the sequence.

sequence_name

Specifies the name of the sequence to be dropped.

Example

The following is an example of the DROP SEQUENCE statement:

```
DROP SEQUENCE pub.customer;
```

Authorization

To delete a sequence, you must have DBA privileges or ownership of the table.

Related Statement

CREATE SEQUENCE

DROP SYNONYM

Drops the specified synonym.

Syntax

```
DROP [ PUBLIC ] SYNONYM synonym ;
```

Parameters

`PUBLIC`

Specifies that the synonym was created with the `PUBLIC` argument.

`SYNONYM` *synonym*

Name for the synonym.

Notes

- If `DROP SYNONYM` specifies `PUBLIC` and the synonym was not a public synonym, SQL generates the "base table not found" error.
- If `DROP SYNONYM` does not specify `PUBLIC` and the synonym was created with the `PUBLIC` argument, SQL generates the "base table not found" error.

Example

The following is an example of the `DROP SYNONYM` statement:

```
DROP SYNONYM customer ;
DROP PUBLIC SYNONYM public_suppliers ;
```

Authorization

Must have `DBA` privilege or ownership of the synonym (for `DROP SYNONYM`).

Related statement

CREATE SYNONYM

DROP TABLE

Deletes the specified table.

If a multi-tenant table is dropped, then, the `DROP TABLE` statement deletes the table partition for every tenant defined in the table.

You can also use the DROP TABLE statement to drop a partitioned table. Dropping a partitioned table is an offline operation. When a partitioned table is dropped, all the table, index, and LOB partitions of the table are deleted. A table cannot be dropped if a partition in the table is marked as a read-only partition. A partitioned table can be dropped only when all its partitions are deallocated and since read-only partitions cannot be deallocated, the table containing them cannot be dropped.

Note: The DROP TABLE statement for partitioned tables is an online operation for partitioned tables in schema other than PUB.

You can use the DROP TABLE statement as an online and offline operation. For further information, see the Notes section below.

Syntax

```
DROP TABLE [owner_name] table_name ;
```

Parameters

owner_name

Specifies the owner of the table.

table_name

Name of the table to be dropped.

Notes

- If *owner_name* is specified and is different from the name of the user executing the statement, then the user must have DBA privileges.
- When a table is dropped, the indexes on the table and the privileges associated with the table are dropped automatically.
- Views dependent on the dropped table are not automatically dropped, but become invalid.
- If the table is part of another table's referential constraint (if the table is named in another table's REFERENCES clause), the DROP TABLE statement fails. You must DROP the referring table first.
- When the DROP TABLE statement is applied to a multi-tenant table, it deletes the table for every tenant defined in the table.
- The DROP TABLE statement is an online operation for the following cases:
  - All data in the Type II area

    Note: If tables are in the type II area but have no indexes created on them, then the DROP TABLE statement for those tables will be an offline operation.

  - Tables that are not multi-tenant
  - Data in schema other than PUB
  - Databases without JTA transactions
  - Tables without OpenEdge Replication

- The DROP TABLE statement is an offline operation for the following cases:
  - Table objects (indexes and LOBs) and tables in areas other than the Type II area
  - Multi-tenant tables
  - Tables in PUBLIC (PUB) schema
  - JTA transactions
  - Tables with OpenEdge Replication

Example

The following is an example of the DROP TABLE statement:

```
DROP TABLE customer ;
```

Authorization

To delete a table, you must have DBA privileges or ownership of the table.

Related statement

CREATE TABLE

DROP TENANT

Deletes a tenant from a multi-tenant database.

Syntax

```
DROP TENANT [owner_name]tenant_name;
```

Parameters

tenant_name

Specifies the name of the tenant you want to delete. The name of the tenant must exist in the database.

owner_name

Specifies the name of the schema which qualifies the table.

Notes

Each database contains users, domains, and tenants. A DBA must follow the order specified below to drop a tenant from the database:

1. DROP USER for every user in the domain.
2. DROP DOMAIN for every domain defined for the tenant.
3. DROP TENANT.

Example

The following example shows how to drop the tenant, `mtcustomer`, from the multi-tenant table:

```
DROP TENANT pub.mtcustomer;
```

If a tenant that is being dropped belongs to one or more groups, it is first disassociated from the group(s) before being dropped.

Authorization

To drop a tenant, you must have DBA privileges.

Related Statement

CREATE TENANT

DROP TRIGGER

Deletes a Trigger.

`DROP TRIGGER` is an online operation. So, this operation can be performed while users are still connected to the database. However, there are two cases where this operation is not online. They are mentioned in the **Exception** section.

Online `DROP TRIGGER` provides high concurrency when compared to other online operations. For example:

- When online `DROP TRIGGER` of a table is being executed in one transaction, users in other transactions are not allowed to perform INSERT/UPDATE/DELETE operations on that table and are allowed to only perform the SELECT operation on that table. There are no restrictions on other tables.

Exception

You cannot perform online operation of `DROP TRIGGER` statement if:

- The table on which trigger is defined is in PUB schema.
- The database is enabled for replication.

Syntax

```
DROP TRIGGER [owner_name.]trigger_name ;
```

Parameters

owner_name

Specifies the owner of the trigger.

trigger_name

Names the trigger to drop.

Example

The following is an example of the DROP TRIGGRER statement:

```
DROP TRIGGER sal_check ;
```

Authorization

Must have DBA privilege or ownership of the trigger.

Related statement

CREATE TRIGGER

DROP USER

Deletes the specified user.

Syntax

```
DROP USER {'username' | 'username@domain_name'};
```

Parameters

username

Specifies the user name to be deleted. The username must be enclosed in single quotes.

domain_name

Specifies the domain name that the user is associated with. The username@domain_name must be enclosed in single quotes.

Notes

In a database with tenants, users, and domains, a DBA must complete the following steps to drop a tenant from the database:

1. DROP USER for every user in the domain.
2. DROP DOMAIN for every domain defined for the tenant. For more information on dropping a user.
3. DROP TENANT.

Example

Dropping a user

In this example, an account with DBA privileges drops the username 'Jasper':

```
DROP USER 'Jasper' ;
```

Dropping a user associated with a domain

The example displayed below shows how to drop a user associated with a domain. An user with DBA privileges deletes the user Jasper associated with the domain name mtdomain:

```
DROP USER 'Jasper@mtdomain';
```

Authorization

Must have DBA privileges.

Related statements

BEGIN-END DECLARE SECTION, CREATE USER

DROP VIEW

Deletes the view from the database.

Syntax

```
DROP VIEW [owner_name.]view_name ;
```

Parameters

owner_name

Specifies the owner of the view.

view_name

Names the view to drop.

Notes

- If *owner_name* is specified and is different from the name of the user executing the statement, then the user must have DBA privileges.
- When a view is dropped, other views that are dependent on this view are not dropped. The dependent views become invalid.

Example

The following is an example of the DROP VIEW statement:

```
DROP VIEW newcustomers ;
```

Authorization

Must have DBA privilege or ownership of the view.

Related statement

CREATE VIEW

GRANT

Grants various privileges to the specified users of the database.

There are two forms of the GRANT statement:

- Grant database-wide privileges, such as system administration (DBA), general creation (RESOURCE), audit administration (AUDIT_ADMIN), audit archive (AUDIT_ARCHIVE), or audit insert (AUDIT_INSERT).
- Grant various privileges on specific tables and views. Privilege definitions are stored in the system tables SYSDBAUTH, SYSTABAUTH, and SYSCOLAUTH for the database, tables, and columns, respectively.

Note: You must use separate commands to grant DBA or RESOURCE privileges with any of the AUDIT privileges. Using the same command to grant a user with DBA or RESOURCE privileges and any of the AUDIT privileges results in an error.

You must use the fully qualified username to grant privileges to tenant data in a multi-tenant table.

Syntax

```
GRANT { RESOURCE, DBA, AUDIT_ADMIN, AUDIT_ARCHIVE, AUDIT_INSERT }
    TO user_identifier [ , user_identifier ] , ...
    [ WITH GRANT OPTION ];
```

Note: The GRANT DBA and GRANT RESOURCE statements do not support the WITH GRANT OPTION syntax.

Parameters

RESOURCE

Allows the specified users to issue CREATE statements.

DBA

Allows the specified users to create, access, modify, or delete any database object, and to grant other users any privileges.

TO username [, *username*] , . . .

Grants the specified privileges on the table or view to the specified list of users.

WITH GRANT OPTION

Allows the specified users to grant their privileges or a subset of their privileges to other users. The WITH GRANT OPTION syntax is not supported by GRANT DBA and GRANT RESOURCE statements.

Syntax for granting privileges to specific tables and views

This is the syntax to grant privileges on specific tables and views:

```
GRANT {privilege[, privilege] , . . . | ALL [ PRIVILEGES]}
 ON table_name
 TO  {username[, username] , . . . |  PUBLIC }
  [ WITH GRANT OPTION ] ;
```

Syntax for the privilege variable

This is the syntax for the privilege variable:

```
{ SELECT | INSERT | DELETE | INDEX
   | UPDATE [ ( column , column , . . . ) ]
   | REFERENCES [ ( column , column , . . . ) ]}
```

Syntax for assigning sequence privileges

Use the following syntax to assign sequence privileges:

```
GRANT [SELECT | UPDATE]
ON SEQUENCE schema.sequence
TO user_name[,user_name]...
```

Parameters

SELECT

Allows specified users to read data from the sequence.

UPDATE

Allows specified users to modify data for the sequence.

Syntax to execute stored Java procedures

The following syntax is a variation on the GRANT statement that enables the user to execute stored Java procedures:

```
GRANT EXECUTE ON StoredJavaProcedureName
 TO  {username[, username] , ... |  PUBLIC }
    [ WITH GRANT OPTION ] ;
```

Parameters

RESOURCE

Allows the specified users to issue CREATE statements.

DBA

Allows the specified users to create, access, modify, or delete any database object, and to grant other users any privileges.

user_identifier

Identifies a username. For a tenant user, you must mention the fully qualified user name, username@domain_name, to grant access to a user.

Uses the following syntax:

```
username | username@domain_name
```

Note:

You must not use commands, such as:

```
GRANT DBA to 'user' WITH GRANT OPTION
```

or:

```
GRANT RESOURCE to 'user' WITH GRANT OPTION
```

The above commands return syntax errors.

`TO username [ , username ] , ...`

Grants the specified privileges on the table or view to the specified list of users.

`SELECT`

Allows the specified users to read data from the table or view.

`INSERT`

Allows the specified users to add new rows to the table or view.

`DELETE`

Allows the specified users to delete rows from the table or view.

`INDEX`

Allows the specified users to create an index on the table or view.

`UPDATE [ ( column , column , ... ) ]`

Allows the specified users to modify existing rows in the table or view. If followed by a column list, the users can modify values only in the columns named.

`REFERENCES [ ( column , column , ... ) ]`

Allows the specified users to refer to the table from other tables' constraint definitions. If followed by a column list, constraint definitions can refer only to the columns named.

For more detail on constraint definitions, see the Column constraints and Table constraints entries of this topic.

ALL

Grants all privileges for the table or view including the ALTER privilege. The ALTER privilege gives the user the ability to ALTER TABLE statements for the given table.

Note: The ALTER privilege cannot currently be granted by itself.

ALTER

Grants the ALTER privilege for the table or view. This privilege is granted as part of the ALL privilege and cannot be granted by itslef.

TO PUBLIC

Grants the specified privileges on the table or view to any user with access to the system.

WITH GRANT OPTION

Allows the specified users to grant their privileges or a subset of their privileges to other users.

Examples

GRANT statement

The following example illustrates the GRANT statement:

```
GRANT DELETE ON cust_view TO dbuser1 ;
GRANT SELECT ON newcustomers TO dbuser2 ;
```

If the *username* specified in a RESOURCE or DBA GRANT operation does not already exist, the GRANT statement creates a row in the SYSDBAUTH system table for the new *username*. This row is not deleted by a subsequent REVOKE operation.

Granting privileges to a user associated with a multi-tenant table

In the following example, a DBA grants DELETE and SELECT privileges to the user dbuser1 associated with domuser1 of a multi-tenant table:

```
GRANT DELETE ON cust_view TO dbuser1@domuser1 ;
GRANT SELECT ON newcustomers TO dbuser2@domuser1 ;
```

The user dbuser1, dbuser2 must be associated with the domain name domuser1 in the multi-tenant table.

For more information on SQL database user authentication and GRANT statement, see *Learn about Identity Management*.

Granting audit administration and audit archive privileges

In this example, audit administration and audit archive privileges are granted to bsmith:

```
GRANT AUDIT_ADMIN, AUDIT_ARCHIVE TO bsmith WITH GRANT OPTION;
```

Because these privileges are granted to bsmith WITH GRANT OPTION, bsmith may now grant these two privileges to other users.

Granting privileges to modify a sequence

In this example, the sequence generator grants user slsadmin the ability to modify the customer number sequence:

```
GRANT UPDATE
     ON SEQUENCE pub.customer_sequence     TO slsadmin;
```

Authorization

To grant permissions for a user and to transfer the permissions from one user to others, you must have DBA privilege, ownership of the table, or all the specified privileges on the table (granted with the WITH GRANT OPTION clause). You must have DBA or AUDIT_ADMIN WITH GRANT privileges to grant auditing privileges.

Related statement

REVOKE

INSERT

Inserts new rows into the specified table or view that will contain either the explicitly specified values or the values returned by the query expression.

When you insert rows in a multi-tenant table, the data is inserted into the appropriate tenant partition. The INSERT statement is supplemented with a TENANT clause for SQL to understand which table partition receives the inserted row.

For a regular tenant in a multi-tenant table, the TENANT clause is optional. However, the tenant name must match the user's tenancy. Otherwise, the INSERT statement returns an error.

For a super-tenant or a DBA in a multi-tenant table, the TENANT clause must be the name of an existing tenant, or a default tenant. This conveys the information to the INSERT statement about the point of insertion of row(s) in an existing or default tenant partition of the multi-tenant table.

When the INSERT statement is applied to the group partition for a tenant, the TENANT clause names the tenant that is part of the multi-tenant table that is being inserted with rows. Any indexes for the table are updated using the partition information for the group.

You can also use the INSERT statement to insert rows into the specified partition of a partitioned table. You must have the write permissions on the partitioned table to add rows to it. Executing the INSERT statement to insert rows into a partitioned table may result in an error in the following cases:

- If all partitioned columns of the partitioned table are not specified in the INSERT statement or if they do not have any default values when the table is created.
- If the specified values of the partitioned columns do not determine in which partition the row should be inserted.
- If there is no space allocated in the partition to which the inserted row belongs.
- If the target partition is a read-only partition.

Syntax

```
INSERT INTO [owner_name] {table_name|view_name}
  [ TENANT tenant_name ]
  [ ( column_name[, column_name] , ... ) ]
    [ ( column_name[, column_name] , ... ) ]
       { VALUES ( value[, value] , ... ) |query_expression} ;
```

Notes

- If the optional list of column names is specified, then only the values for those columns are required. The rest of the columns of the inserted row will contain NULL values, provided that the table definition allows NULL values and there is no DEFAULT clause for the columns. If a DEFAULT clause is specified for a column and the column name is not present in the optional column list, then the column is given the default value.
- If the optional list is not specified, then the column values must be either explicitly specified or returned by the query expression. The order of the values should be the same as the order in which the columns are declared in the declaration of the table or view.
- The VALUES (...) form for specifying the column values inserts one row into the table. The query expression form inserts all the rows from the query results.
- A SELECT statement utilizing a NOLOCK hint can be used within an INSERT statement.

  For example:

  ```
  INSERT INTO PUB.CUSTOMER
  SELECT * FROM  PUB.ARCHIVE_CUST
   WHERE ...
   WITH (NOLOCK);
  ```

  For more information using the NOLOCK hint in a SELECT statement, see .
- can be used if If the table contains a foreign key and there is no corresponding primary key that matches the values of the foreign key in the record being inserted, then the insert operation is rejected.

Examples

INSERT statement

The following provides examples of the INSERT statement:

```
INSERT INTO customer (cust_no, name, street, city, state)
     VALUES (1001, 'RALPH', '#10 Columbia Street', 'New York', 'NY') ;

INSERT INTO neworders (order_no, product, qty)
     SELECT order_no, product, qty
          FROM orders
          WHERE order_date = SYSDATE ;
```

Inserting a row in the tenant partition of a multi-tenant table

The example below directs the INSERT statement to insert a new row in the tenant partition for SNCSoftware in the multi-tenant table mtcustomer.

```
INSERT INTO pub.mtcustomer TENANT SNCSoftware
(custnum, name)
 VALUES (9999, 'West Side Sports');
```

Inserting rows from the default partition to other tenants

The example below lists two INSERT statements that move rows from the DEFAULT partition of mtcustomer, and then distribute the rows with an even customer number to the tenant SNCSoftware and rows with an odd customer number to the tenant OEDProducts.

```
INSERT INTO pub.mtcustomer TENANT SNCSoftware
 SELECT *
  FROM pub.mtcustomer AS mtc
  WHERE tenantName_tbl (mtc) = 'default'
  AND MOD (custnum, 2) = 0;

INSERT INTO pub.mtcustomer TENANT OEDProducts
 SELECT *
  FROM pub.mtcustomer AS mtc
  WHERE tenantName_tbl (mtc) = 'default'
  AND MOD (custnum, 2) <> 0;

DELETE
 FROM pub.mtcustomer AS mtc
 WHERE tenantName_tbl (mtc) = 'default';
```

Inserting rows into a table partitioned by LIST

The examples below insert rows into the table Pub.tpcustomer that is partitioned by LIST.

Assume that the table is partitioned based on SalesRep as given below:

- PART1_LIST SalesRep IN ('SLS')
- PART2_LIST SalesRep IN ('JLP')
- PART3_LIST SalesRep IN ('KIK')
- PART4_LIST SalesRep IN ('BBB')

The below example inserts rows into the partition PART1_LIST:

```
INSERT INTO Pub.tpcustomer (cust_num, SalesRep) VALUES ( 100, 'SLS' );
```

The below example inserts rows into the partition PART4_LIST:

```
INSERT INTO Pub.tpcustomer (cust_num, SalesRep) VALUES ( 101, 'BBB' );
```

The below example returns an error since the row being inserted does not belong to any partition. Here the value of the SalesRep column does not determine any specific partition.

```
INSERT INTO Pub.tpcustomer (cust_num, SalesRep) VALUES ( 102, 'XYZ' );
```

Inserting rows into a table partitioned by RANGE

The examples below insert rows into the table Pub.tporder that is partitioned by RANGE.

Assume that the table is partitioned based on OrderDate as given below:

- PART1_RANGE OrderDate <= ('01/01/1998')
- PART2_RANGE OrderDate <= ('01/01/2010')
- PART3_RANGE OrderDate <= ('01/01/2018')

The below example inserts rows into the partition PART1_RANGE:

```
INSERT INTO Pub.tporder (ordernum, custnum, OrderDate) VALUES ( 1, 100, '10/10/1990'
 );
```

The below example inserts rows into the partition PART3_RANGE:

```
INSERT INTO Pub.tporder (ordernum, custnum, OrderDate) VALUES ( 2, 101, '10/10/2017'
 );
```

The below example inserts rows into the partition PART3_RANGE:

```
INSERT INTO Pub.tporder (ordernum, custnum, OrderDate) VALUES ( 2, 101, '01/01/2018'
 );
```

The below example returns an error since the row being inserted does not belong to any partition.

```
INSERT INTO Pub.tporder (ordernum, custnum, OrderDate) VALUES ( 2, 101, '01/01/2019'
 );
```

Authorization

To insert a row, you must have DBA privileges, ownership of the table, INSERT privilege on the table, or SELECT privilege on all the tables or views referred to in the *query_expression*, if it is specified.

A user mapped to a super-tenant can view data for all the tenants only if the user has the required permissions on the tables referenced in the statement.

Related statement

REVOKE

LOCK TABLE

Explicitly locks one or more specified tables for shared or exclusive access.

Syntax

```
LOCK TABLE table_name[ , table_name] , ...
  IN { SHARE | EXCLUSIVE } MODE ;
```

Parameters

table_name

The table in the database that you want to lock explicitly. You can specify one table or a comma-separated list of tables.

SHARE MODE

Allows all transactions to read the tables. Prohibits all **other** transactions from modifying the tables. After you acquire an explicit lock on a table in SHARE MODE, any SELECT statements in your transaction can read rows and **do not** implicitly acquire individual record locks. Any INSERT, UPDATE, and DELETE statements **do** acquire record locks.

EXCLUSIVE MODE

Allows the current transaction to read and modify the tables, and prohibits any other transactions from reading or modifying the tables. After you acquire an explicit lock on a table in EXCLUSIVE MODE, you can SELECT, INSERT, UPDATE, and DELETE rows, and your transaction **does not** implicitly acquire individual record locks for these operations.

Notes

- The LOCK TABLE statement might encounter a locking conflict with another transaction.
- The SHARE MODE option detects a locking conflict if another transaction:
  - Locked the table in EXCLUSIVE MODE and has not issued a COMMIT or ROLLBACK
  - Inserted, updated, or deleted rows in the table and has not issued a COMMIT or ROLLBACK

- The EXCLUSIVE MODE option detects a locking conflict if another transaction:
  - Locked the table in SHARE MODE or EXCLUSIVE MODE and has not issued a COMMIT or ROLLBACK
  - Read from, inserted, updated, or deleted rows and has not issued a COMMIT or ROLLBACK
- When there is a locking conflict, the transaction is suspended and the database returns an error. You might configure the time at which the transaction is suspended. The default is five seconds.
- You can use explicit table locking to improve the performance of a single transaction, at the cost of decreasing the concurrency of the system and potentially blocking other transactions. It is more efficient to lock a table explicitly if you know that the transaction will be updating a substantial part of a table. You gain efficiency by decreasing the overhead of the implicit locking mechanism, and by decreasing any potential wait time for acquiring individual record locks on the table.
- You can use explicit table locking to minimize potential deadlocks in situations where a transaction is modifying a substantial part of a table. Before making a choice between explicit or implicit locking, compare the benefits of table locking with the disadvantages of losing concurrency.
- The database releases explicit and implicit locks only when the transaction ends with a COMMIT or ROLLBACK operation.

Example

Unless another transaction holds an EXCLUSIVE lock on the teratab and megatab tables, the SHARE MODE example explicitly locks the tables. The shared lock allows all transactions to read the tables. Only the current transaction can modify the tables, as shown in the following example:

```
LOCK TABLE teratab, megatab IN SHARE MODE ;
```

Unless another transaction holds a lock on the teratab table, the EXCLUSIVE MODE example locks the teratab table for exclusive use by the current transaction. No other transactions can read or modify the teratab table, as shown in the following example:

```
LOCK TABLE teratab IN EXCLUSIVE MODE ;
```

Without a table lock, the first SELECT statement in the following example could exceed the limits of the record lock table, while the LOCK TABLE statement prevents the subsequent SELECT statement from consuming the record lock table:

```
-- Without a table lock, this SELECT statement creates an
-- entry in the record lock table for every row in teratab.
 SELECT COUNT (*) FROM teratab ;
 -- The LOCK TABLE IN SHARE MODE operation preserves the
-- record lock table resource.
 LOCK TABLE teratab IN SHARE MODE ;
SELECT COUNT (*) FROM teratab ;
```

Authorization

Must have DBA privilege or SELECT privilege on the table.

Related statements

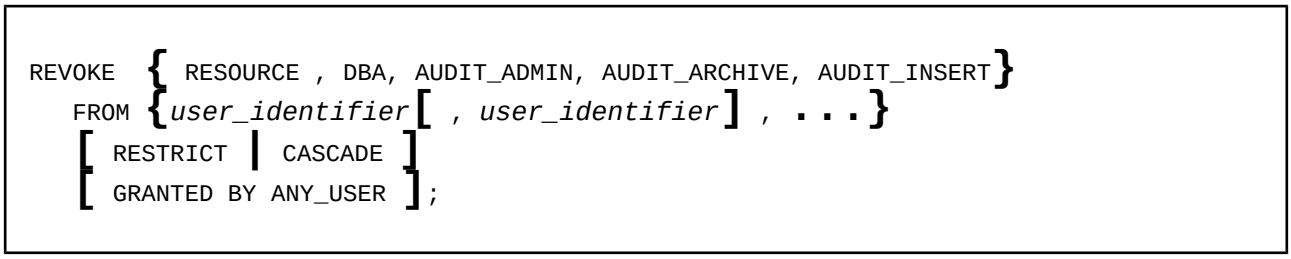
COMMIT, ROLLBACK, SET TRANSACTION ISOLATION LEVEL

REVOKE

Revokes various privileges from the specified users of the database. There are two forms of the REVOKE statement:

- Revoke database-wide privileges, either system administration (DBA), general creation (RESOURCE), audit administration (AUDIT_ADMIN), audit archive (AUDIT_ARCHIVE), or audit insert (AUDIT_INSERT)
- Revoke various privileges on specific tables and views, sequences, stored procedures and user defined functions.

Syntax



Parameters

RESOURCE

Revokes the privilege to issue CREATE statement for specified users.

DBA

Revokes the privilege to create, access, modify, or delete any database object, and revokes the privilege to grant other users any privileges for specified users.

AUDIT_ADMIN

Revokes the privilege to administer and maintain a database auditing system for specified users.

AUDIT_ARCHIVE

Revokes the privilege to read and delete audit records for specific users.

AUDIT_INSERT

Revokes the privilege to insert application audit records for specified users.

FROM *user_identifier* [, *user_identifier*] , ...

Revokes the specified privileges on the table or view from the specified list of users.

EXECUTE

Revokes the EXECUTE privileges for the specified list of users from using user defined functions and stored procedures.

RESTRICT | CASCADE

Prompts SQL to check if the privilege being revoked was passed onto other users. If the original privilege was included the WITH GRANT OPTION clause, then the REVOKE statement fails and generates an error. If the privilege was not passed on, the REVOKE statement succeeds.

If the REVOKE statement specifies CASCADE, revoking the access privileges from a user also revokes the privileges from all users who received the privilege from that user.

If the REVOKE statement does not specify either RESTRICT or CASCADE, then the behavior follows only CASCADE.

user_identifier

Identifies a username. For a tenant user, you must mention the fully qualified user name, username@domain_name, to revoke access from a user.

Uses the following syntax:

```
username | username@domain_name
```

Note: CASCADE is not supported for AUDIT_ADMIN, AUDIT_ARCHIVE, and AUDIT_INSERT privileges. The only user who can revoke an audit privilege is the user who granted it.

GRANTED BY ANY_USER

A DBA can use this phrase to revoke all access privileges to a table for a specified user, even if the user was the creator of the table. This phrase is only applicable to users with DBA privileges.

Syntax to revoke privileges on stored procedures

Use the following syntax to revoke privileges for user(s) on specified procedures:

```
REVOKE EXECUTE ON StoredJavaProcedureName
 FROM  {username[, username] , ... |  PUBLIC }
   [ RESTRICT | CASCADE ];
```

Syntax to revoke sequence privileges

Use the following syntax to revoke sequence privileges:

```
REVOKE [SELECT | UPDATE]
ON SEQUENCE schema.sequence
FROM user_name[,user_name]...
```

Parameters

SELECT

For the specified users, removes the privilege to read data from the sequence.

UPDATE

For the specified users, removes the privilege to modify data for the sequence.

Examples

In this example, the audit administration privilege is revoked from bsmith:

```
REVOKE AUDIT_ADMIN FROM bsmith RESTRICT;
```

Revoking privileges to modify a sequence

In this example, the UPDATE permission of the sequence pub.customer_sequence is revoked from the user slsadmin:

```
REVOKE UPDATE
     ON SEQUENCE pub.customer_sequence     FROM slsadmin;
```

Syntax to revoke privileges on specific tables and views

The syntax to revoke privileges on specific tables and views is shown below:

Syntax

```
REVOKE [ GRANT OPTION FOR ]
  {privilege[, privilege] , ... | ALL [ PRIVILEGES ]}
   ON table_name
   FROM {user_identifier [ , user_identifier] , ... | PUBLIC }
        [ RESTRICT | CASCADE ]
   GRANTED BY ANY_USER;
```

Parameters

GRANT OPTION FOR

Revokes the GRANT option for the privilege from the specified users. The actual privilege itself is not revoked. If specified with RESTRICT, and the privilege is passed on to other users, the REVOKE statement fails and generates an error. Otherwise, GRANT OPTION FOR implicitly revokes any privilege the user might have provided to other users.

privilege

The syntax for the privilege item is shown below:

```
{ SELECT | INSERT | DELETE | INDEX
  | UPDATE [ ( column , column , ... ) ]
    | REFERENCES [ ( column , column , ... ) ]} ;
```

privilege [, *privilege*] , ... | ALL [PRIVILEGES]

List of privileges to be revoked. Refer to the description in the GRANT statement. Revoking RESOURCE and DBA privileges can only be done by the administrator or a user with DBA privileges.

If more than one user grants access to the same table to a user, then all the grantors must perform a revoke for the user to lose access to the table.

Using the keyword ALL revokes all the privileges granted on the table or view.

FROM PUBLIC

Revokes the specified privileges on the table or view from any user with access to the system.

RESTRICT | CASCADE

Prompts SQL to check to see if the privilege being revoked was passed on to other users. If the original privilege included the clause WITH GRANT OPTION, then the REVOKE statement fails and generates an error. If the privilege was not passed on, the REVOKE statement succeeds.

If the REVOKE statement specifies CASCADE, revoking the access privileges from a user also revokes the privileges from all users who received the privilege from that user.

If the REVOKE statement does not specify either RESTRICT nor CASCADE, the behavior follows only CASCADE.

GRANTED BY ANY_USER

A DBA can use this phrase to revoke all or specific access privileges to a table for a specified user, even if the user was the creator of the table. This phrase is only applicable to users with DBA privileges.

Examples

Revoking privileges

In the following example, REVOKE is used on INSERT and DELETE privileges:

```
REVOKE INSERT ON customer FROM dbuser1;
REVOKE DELETE ON cust_view FROM dbuser2;
```

If the *username* specified in a GRANT DBA or GRANT RESOURCE operation does not already exist in the authorization tables, the GRANT statement creates a row in the SYSDBAUTH system table for the new *username*. This row is not deleted by a subsequent REVOKE operation.

Revoking privileges from a user associated with the domain of multi-tenant table

In the following example, a DBA revokes privileges from dbuser1 associated with domuser1 domain of a multi-tenant table:

```
REVOKE INSERT ON customer FROM dbuser1@domuser1;
REVOKE DELETE ON cust_view FROM dbuser2@domuser1;
```

The user names dbuser1 and dbuser2 must be associated with the domain name domuser1 in the multi-tenant table.

Revoking privileges for a specified user who has been granted privileges by ANY_USER.

In this example, all privileges granted by any_user on PUB.EMPLOYEE table are revoked from bsmith:

```
REVOKE ALL on PUB.EMPLOYEE FROM 'bsmith' GRANTED BY ANY_USER;
```

In this example, only SELECT and INSERT privileges granted by any_user on PUB.EMPLOYEE table are revoked from bsmith:

```
REVOKE SELECT, INSERT on PUB.EMPLOYEE FROM 'bsmith' GRANTED BY ANY_USER;
```

For more information on SQL database user authentication and REVOKE statement, see *Learn about Identity Management*.

Authorization

To revoke privileges, you must have DBA privilege or ownership of the table. To revoke audit privileges, the user must have DBA or AUDIT ADMINISTRATION WITH GRANT privileges **and** you must be the user who granted the audit privilege.

Related statement

GRANT

ROLLBACK

Ends the current transaction and undoes any database changes performed during the transaction.

Syntax

```
ROLLBACK [ WORK ] ;
```

Rules to use the ROLLBACK statement

The following rules should be followed while using the ROLLBACK statement:

- It can be used in an embedded SQL application.
- It can be used in tools like SQL Explorer which expose the COMMIT operation as a SQL statement.
- It cannot be used in an ODBC or JDBC application because ODBC exposes the COMMIT operation as the ODBC API SQLEndTran() and JDBC exposes the COMMIT operation in the connection object as the commit() method.
- It cannot be used in an OE SQL stored procedure because a stored procedure runs on the OpenEdge SQL server while transactions can be controlled only from a SQLclient.

Notes

- Under certain circumstances, SQL marks a transaction for abort but does not actually roll it back immediately. Without an explicit `ROLLBACK`, any subsequent updates do not take effect. A `COMMIT` statement causes SQL to recognize the transaction as marked for abort and instead implicitly rolls back the transaction.
- SQL marks a transaction for abort in the event of a hardware or software system failure. This transaction is rolled back during recovery.

Authorization

None

Related statements

COMMIT

SELECT

Selects the specified column values from one or more rows contained in the tables or views specified in the query expression. The selection of rows is restricted by the `WHERE` clause. You can also use clauses like `GROUP BY`, `HAVING`, `ORDER BY` with the `SELECT` statement. These clauses are described in detail in the further sections. To limit the number of rows that are returned, you can implement a query paging solution using the `OFFSET` and `FETCH` clauses. The temporary table derived through the clauses of a select statement is called a result table.

When selecting rows in a multi-tenant table, a regular tenant can only view the rows in its partition, but a DBA or a super-tenant can view all the tenant partitions in the multi-tenant tables being accessed.

A super-tenant can also view a list of tenant IDs and tenant names that are associated with data from a tenant partition or from a group partition, by using the `tenantid_tbl()` and the `tenantName_tbl()` functions, respectively.

These functions logically evaluate to the set of IDs and names of the tenants that are associated with a tenant partition or with a group partition.They also enable the query to specify the tenants for which data should be selected.

When selecting rows from a partitioned table, the `SELECT` statement returns data from all the partitions created for the table as determined by the predicates (restrictions) in the statement.

Syntax

```
SELECT [ ALL | DISTINCT ] [TOP n]
  { *
    |{table_name|alias} * [ , {table_name| alias} * ]...
       | expr [[ AS ][ ' ] column_title [ ' ]]
          [,  expr [[ AS ][ ' ] column_title [' ]]]...
  }
FROM table_ref [, table_ref]...[{ NO REORDER }] [ WITH (NOLOCK )]
      [ WHERE search_condition][ GROUP BY [ table ]column_name
         [,[table]column_name ]...
      [HAVING search_condition];

  [ORDER BY ordering_condition]
  [OFFSET offset_value {ROW | ROWS }
    [FETCH {FIRST | NEXT}fetch_value {ROW | ROWS} ONLY ]]
  [WITH locking_hints]
  [FOR UPDATE update_condition];
```

Note: The SELECT statement does not the support the use of the FOR UPDATE caluse and the ORDER BY clause or any aggreagte function in the same staement. For example, the statement SELECT * FROM table ORDER BY field1 FOR UPDATE; will return an error.

Parameters

column_list

See the .

TOP *nsearch_condition*

See the .

FROM *table_list*

See the .

WHERE *search_condition*

See the .

GROUP BY *grouping_condition*

See the .

HAVING *search_condition*

See the .

`ORDER BY` *ordering_condition*

See the .

`OFFSET` *offset_value*

See the .

`FETCH { FIRST | NEXT }`

See the .WITH *locking_hints*

See the .

`FOR UPDATE` *update_condition*

See the .

Example

The following example displays the different forms of the SELECT statement:

The example below illustrates the use of SELECT statement.

```
SELECT * FROM pub.customer;
```

Assuming the user is mapped to a DBA or a super-tenant, the example below directs the SELECT statement to join three multi-tenant tables.

When joining three multi-tenant tables, the DBA or the super-tenant must make sure that data from one tenant, `pub.mtcustomer`, is joined only with the data owned by the same tenant in the other tables, `pub.mtorder` and `pub.mtorderline`; especially when primary keys, `custnum` and `ordernum`, are unique for a tenant, but not unique across all the tenants.

```
SELECT
    tenantName_tbl (c) as ten_name,
    c.name as c_name,
    COUNT (*)
FROM
    Pub.mtcustomer AS c
    INNER JOIN pub.mtorder AS o
    ON tenantId_tbl (c) = tenantId_tbl (o)
    AND c.custnum = o.custnum
    INNER JOIN pub.mtorderline AS ol
    ON tenantId_tbl (o) = tenantId_tbl (ol)
    AND o.ordernum = ol.ordernum
GROUP BY
    ten_name, c_name;
```

COLUMN_LIST clause

Specifies which columns to retrieve by the SELECT statement.

Syntax

```
[ ALL | DISTINCT ]
   { * |{table_name|alias.}* [ , {table_name.|alias.} * ]...
          |expr[[ AS ][ ' ]column_title[ ' ]]
                   [, expr[[ AS ][ ' ]column_alias[ ' ]]]...
          |[table|alias.]column_name , ...]
   }
```

Parameters

[ALL | DISTINCT]

Indicates whether a result table omits duplicate rows. ALL is the default and specifies that the result table includes all rows. DISTINCT specifies that a table omits duplicate rows.

*** |{*table_name*. |*alias*. } ***

Specifies that the result table includes all columns from all tables named in the FROM clause.

*** *expr*[[AS][']*column_alias*[']]**

Specifies a list of expressions, called a select list, whose results will form columns of the result table. Typically, the expression is a column name from a table named in the FROM clause. The expression can also be any supported mathematical expression, scalar function, or aggregate function that returns a value.

The optional *column_alias* argument specifies a new heading for the associated column in the result table. You can also use the *column_title* in an ORDER BY clause. Enclose the new title in single or double quotation marks if it contains spaces or other special characters, including hyphens.

Note: A table alias cannot be used to qualify a column alias. A column alias can only be used without a qualifier because it is not a part of any table definition.

[*table* | *alias*.]*column_name* , ...]

Specifies a list columns from a particular table or alias.

Example

Both these statements return all the columns in the customer table to the select list:

```
SELECT * FROM Customer;
SELECT Customer.* FROM Customer;
```

The *table_name*.* syntax is useful when the select list refers to columns in multiple tables and you want to specify all the columns in one of those tables. For example:

```
SELECT Customer.CustNum, Customer.Name, Invoice.*
     FROM Customer, Invoice ;
```

The following example illustrates using the *column_alias* option to change the name of the column:

```
-- Illustrate optional 'column_title' syntax
SELECT
        FirstName AS 'First Name',
        LastName AS 'Last Name',
        state AS 'New England State'
FROM Employee
        WHERE state = 'NH' OR state = 'ME' OR state = 'MA'
OR state = 'VT' OR state = 'CT' OR state = 'RI';
 First Name       Last Name        New England State
--------------- ---------------- ------------------
Justine          Smith            MA
Andy             Davis            MA
Marcy            Adams            MA
Larry            Dawsen           MA
John             Burton           NH
Mark             Hall             NH
Stacey           Smith            MA
Scott            Abbott           MA
Meredith         White            NH
Heather          White            NH
```

You must qualify a column name if it occurs in more than one table specified in the FROM clause, as shown:

```
SELECT Customer.CustNum FROM Customer;
```

```
-- Table name qualifier required
-- Customer table has city and state columns
-- Billto table has city and state columns
 SELECT
        Customer.CustNum,
        Customer.City AS 'Customer City',
        Customer.State AS 'Customer State',
        Billto.City AS 'Bill City',
        Billto.State AS 'Bill State'
FROM Customer, Billto
        WHERE Customer.City = 'Clinton';
  CustNum Customer City    Customer State   Bill City        Bill State
-------- ---------------- ---------------- ------------------------------
  1272     Clinton          MS               Montgomery         AL
  1272     Clinton          MS               Atlanta            GA
  1421     Clinton          SC               Montgomery         AL
  1421     Clinton          SC               Atlanta            GA
  1489     Clinton          OK               Montgomery         AL
  1489     Clinton          OK               Atlanta            GA
```

When there is a conflict between a SELECT list alias and an actual database column, OpenEdge SQL interprets the reference as the database column. Note the following example:

```
SELECT substring (state, 1, 2) state, sum (balance)
FROM pub.customer
GROUP BY state;
```

In the above query, state is ambiguous because it can refer to either database column pub.customer.state or the result of the substring scalar function in the SELECT list. The ANSI standard requires that state refers unambiguously to the database column, therefore, the query groups the result by the database column. The same principle holds true for ambiguous references that appear in WHERE, ON, and HAVING clauses.

TOP clause

Limits the rows returned by an OpenEdge SQL query at the statement level and is supported in subqueries.

Syntax

```
TOP n [ column_name [, column_name ] ..... | * ]
```

Parameters

n

Indicates the number of records per table.

[column_name [, column_name] ..... | *]

Indicates the columns within a table.

When the TOP clause is specified, the OpenEdge SQL server returns the maximum number of rows specified in the clause. The maximum number allowed for the TOP clause is 2,147,483,647.

Example

In the following example, the SELECT statement returns the names of the five customers with the highest account balance:

```
SELECT TOP 5 * FROM pub.customer
     ORDER BY balance DESC;
```

The TOP clause is only allowed in a top-level SELECT statement. Therefore, the TOP clause cannot be used in the following instances:

- When derived tables are used in the query
- Within the CREATE TABLE, CREATE VIEW, UPDATE, and INSERT statements
- In queries used with set operators such as UNION, INTERSECT, and MINUS

In instances when the server performs aggregation on the result set (i.e., through an aggregate function such as SUM or MAX, a GROUP BY clause, or the DISTINCT keyword) the TOP clause should be interpreted as being applied last. When there is no aggregation in the SELECT statement and the result set is also sorted, then SQL will optimize sorting in order to increase query performance.

SELECT TOP is the functional equivalent of the Oracle ROWNUM functionality. Note that SELECT TOP is defined simply in terms of a limit on the result set size, and the optimizer determines how to use this limit for best data access. Thus, SELECT TOP does not have all the "procedural rules" used to define the meaning of the Oracle ROWNUM phrase.

FROM clause

Specifies one or more table references. Each table reference resolves to one table (either a table stored in the database or a virtual table resulting from processing the table reference) whose rows the query expression uses to create the result table.

Syntax

```
FROM table_ref[, table_ref]...[{ NO REORDER }]
```

Parameters

table_ref

There are three forms of table references:

- A direct reference to a table, view, or synonym
- A derived table specified by a query expression in the **FROM** clause
- A joined table that combines rows and columns from multiple tables

If there are multiple table references, SQL joins the tables to form an intermediate result table that is used as the basis for evaluating all other clauses in the query expression. That intermediate result table is the Cartesian product of rows in the tables in the FROM clause, formed by concatenating every row of every table with all other rows in all tables, as shown in the following syntax:

```
table_name[ AS ][alias[ ( column_alias[...] ) ]]
   | ( query_expression ) [ AS ]alias[ (column_alias[...] ) ]
   |[ ( ]joined_table[ ) ]
```

```
FROM table_name[ AS ][alias[ ( column_alias [...] ) ]]
```

Explicitly names a table. The name can be a table name, a view name, or a synonym.

alias

A name used to qualify column names in other parts of the query expression. Aliases are also called correlation names.

If you specify an alias, you must use it, and not the table name, to qualify column names that refer to the table. Query expressions that join a table with itself must use aliases to distinguish between references to column names.

Similar to table aliases, the *column_alias* provides an alternative name to use in column references elsewhere in the query expression. If you specify column aliases, you must specify them for all the columns in *table_name*. Also, if you specify column aliases in the FROM clause, you must use them, and not the column names, in references to the columns.

FROM (*query_expression*) [AS] [*alias* [(*column_alias* [. . .])]]

Specifies a derived table through a query expression. With derived tables, you must specify an alias to identify the derived table.

Derived tables can also specify column aliases. Column aliases provide alternative names to use in column references elsewhere in the query expression. If you specify column aliases, you must specify them for all the columns in the result table of the query expression. Also, if you specify column aliases in the FROM clause, you must use them, and not the column names, in references to the columns.

FROM [( ] *joined_table* [ )]

Combines data from two table references by specifying a join condition, as shown in the following syntax:

```
{ table_ref CROSS JOIN table_ref
  | table_ref [ INNER | LEFT [ OUTER ] ] JOIN
    table_ref ON search_condition
}
```

The syntax currently allowed in the FROM clause supports only a subset of possible join conditions:

- CROSS JOIN specifies a Cartesian product of rows in the two tables. Every row in one table is joined to every row in the other table.
- INNER JOIN specifies an inner join using the supplied search condition.
- LEFT OUTER JOIN specifies a left outer join using the supplied search condition.
- LEFT JOIN specifies the same conditions as an inner join.

You can also specify these and other join conditions in the WHERE clause of a query expression.

{ NO REORDER }

Disables join order optimization for the FROM clause. Use NO REORDER when you choose to override the join order chosen by the optimizer. The braces are part of the syntax for this optional clause.

[WITH (NOLOCK)]

Allows a dirty read to occur in the event records are locked by another user.

Example

For customers with orders, retrieve their names and order info, as shown in the following example:

```
SELECT Customer.CustNum,Customer.Name, Order.OrderNum,Order.OrderDate
     FROM Customer, Order
     WHERE Customer.CustNum = Order.CustNum;
```

WHERE clause

Specifies a search condition that applies conditions to restrict the number of rows in the result table. If the query expression does not specify a WHERE clause, the result table includes all the rows of the specified table reference in the FROM clause.

Syntax

```
WHERE search_condition
```

Parameter

search_condition

Applied to each row of the result table set of the FROM clause. Only rows that satisfy the conditions become part of the result table. If the result of the *search_condition* is NULL for a row, the row is not selected. Search conditions can specify different conditions for joining two or more tables.

Example

For customers with orders, retrieve their names and order info:

```
SELECT Name, City, State
     FROM Customer
     WHERE State = 'NM' ;
```

GROUP BY clause

Specifies grouping of rows in the result table. The results may be grouped by column, alias or expression.

Grouping by column

The result set of a query may be ordered by one or more columns specified in the GROUP BY clause.

Syntax

```
GROUP BY [table_name.]column_name...
```

Notes

- For the first column specified in the GROUP BY clause, SQL arranges rows of the result table into groups whose rows all have the same values for the specified column.
- If you specify a second GROUP BY column, SQL groups rows in each main group by values of the second column.
- SQL groups rows for values in additional GROUP BY columns in a similar fashion.
- All columns named in the GROUP BY clause must also be in the select list of the query expression. Conversely, columns in the select list must also be in the GROUP BY clause or be part of an aggregate function.

Example

This example retrieves name and order info for customers with orders:

```
SELECT DeptCode, LastName
     FROM Employee
     GROUP BY DeptCode;
```

Grouping by alias

The GROUP BY clause orders the result set according to an alias specified in the SELECT statement.

Syntax

```
GROUP BY [alias]...
```

Notes

In this instance, the alias may be used as a simple column reference to a database table, or an actual expression composed of arithmetic expressions, character operators, date operators, or scalar functions. The alias is essentially an alternate name.

Example

In the following example, the GROUP BY clause refers to the "CityState" phrase of the SELECT statement:

```
SELECT CONCAT (State, City) AS "CityState",
       COUNT (city)
  FROM Pub.Customer
  GROUP BY "CityState";
```

Grouping by expression

The GROUP BY clause orders the result set according to an expression used in the SELECT statement.

Syntax

```
GROUP BY [expression]...
```

Notes

The GROUP BY clause can contain any scalar expression which produces a value that is used as a grouping key. An individual column, when it is part of a larger expression in a GROUP BY list, cannot by itself be referenced in the SELECT list. Only the entire expression, which is the grouping key, can be used in the statement's SELECT list. Note that a GROUP BY expression cannot contain an aggregate expression such as SUM.

The GROUP BY clause does not support set differencing operations such as MINUS and INTERSECT.

Example

In the following example, the GROUP BY clause refers to the concatenation expression used in the SELECT statement:

```
SELECT CONCAT (State, City),
       COUNT (city)
  FROM Pub.Customer
  GROUP BY CONCAT (State, City);
```

HAVING clause

Allows you to set conditions on the groups returned by the GROUP BY clause. If the HAVING clause is used without the GROUP BY clause, the implicit group against which the search condition is evaluated is all the rows returned by the WHERE clause.

Syntax

```
HAVING search_condition
```

Notes

A condition of the HAVING clause can compare one aggregate function value with another aggregate function value or a constant.

Example

The HAVING clause in the following example compares the value of an aggregate function

(COUNT (*)) to a constant (10):

```
SELECT CustNum, COUNT(*)
     FROM Order
     WHERE OrderDate < TO_DATE ('3/31/2004')
     GROUP BY CustNum
     HAVING COUNT (*) > 10 ;
```

The query returns the customer number and number of orders for all customers who had more than 10 orders before March 31.

ORDER BY clause

Allows ordering of the rows selected by the SELECT statement. Unless an ORDER BY clause is specified, the rows of the result set might be returned in an unpredictable order as determined by the access paths chosen and other decisions made by the query optimizer. The decisions made will be affected by the statistics generated from table and index data examined by the UPDATE STATISTICS command.

Syntax

```
ORDER BY {expr | posn}[ ASC | DESC ]
     [ , { expr | posn}[ ASC | DESC ] , ...]
```

Parameters

expr

Expression of one or more columns of the tables specified in the FROM clause of the SELECT statement.

posn

Integer column position of the columns selected by the SELECT statement.

ASC | DESC

Indicates whether to order by ascending order (ASC) or descending order. The default is ASC.

Notes

- The ORDER BY clause, if specified, should follow all other clauses of the SELECT statement.
- The selected rows are ordered on the basis of the first *expr* or *posn*. If the values are the same, then the second *expr* or *posn* is used in the ordering.
- A query expression can be followed by an optional ORDER BY clause. If the query expression contains set operators (such as, UNION), then the ORDER BY clause can specify column names, aliases, and positions in the following ways:
  - An alias from the first SELECT statement of the set operator query expression can be used as the ORDER BY key. In this case, the alias indicates the ordering based on the corresponding column from all the query expressions (SELECT statements) in the set operator.
  - A column name from the first SELECT statement of the set operator query expression can be explicitly used as an ORDER BY key. In this case, the column name indicates the ordering based on the corresponding column from all the query expressions (SELECT statements) in the set operator.
  - An ordinal position number can be specified as an ORDER BY key.
- A query expression is considered invalid if the ORDER BY clause:
  - Has an expression

- Refers to a column name or alias that is not from the first SELECT statement
- Has a position number that is greater than the number of columns projected

Examples

ORDER BY clause

The following example demonstrates the ORDER BY clause in the SELECT statement:

```
-- Produce a list of customers sorted by name.
SELECT Name, Address, City, State, PostalCode
     FROM Customer
     ORDER BY Name ;
```

ORDER BY clause in query expressions containing set operators

The following examples demonstrate the ORDER BY clause in query expressions containing set operators:

```
-- Produce a merged list of customers and suppliers with the column position specifying
 the sort key.
SELECT Name, Address, State, PostalCode
     FROM Customer
     UNION
     SELECT Name, Address, State, PostalCode
     FROM Supplier
     ORDER BY 1;
-- Produce a merged list of customers and suppliers sorted by name.
SELECT Name, Address, State, PostalCode
     FROM Customer
     UNION
     SELECT Name, Address, State, PostalCode
     FROM Supplier
     ORDER BY Name;
-- Produce a merged list of customers and suppliers sorted by name.
SELECT cust_name as Name, cust_address, cust_state, cust_postalcode
     FROM Customer
     UNION
     SELECT sup_name, sup_address, sup_state, sup_postalcode
     FROM Supplier
     ORDER BY Name;
```

OFFSET and FETCH clauses

The OFFSET clause specifies the number of rows to skip, before starting to return rows from the query expression. The FETCH clause specifies the number of rows to return, after processing the OFFSET clause. The OFFSET and FETCH clauses are now supported in subqueries.

Syntax

```
[ORDER BY {expr|posn}[ ASC | DESC ]
  [ , { expr|posn}[ ASC | DESC ] , ...]]
  [OFFSET offset_value {ROW | ROWS }
  [FETCH {FIRST | NEXT}fetch_value {ROW | ROWS} ONLY ]]
```

Note: You can use ROW and ROWS interchangeably. Similarly, FIRST and NEXT can be used interchangeably.

Parameters

expr

Expression of one or more columns of the tables specified in the FROM clause of the SELECT statement.

posn

Integer column position of the columns selected by the SELECT statement.

ASC | DESC

Indicates whether to order by ascending order (ASC) or descending order. The default is ASC.

offset_value

Specifies the number of rows to skip, before starting to return the result rows of the query. This value must be a constant: literal or parameter. It does not support expressions or self-contained subqueries. The value must not be negative and the maximum number allowed is 9,223,372,036,854,775,807.

fetch_value

Specifies the number of rows to return, after processing the OFFSET clause. This value must be a constant: literal or parameter. It does not support expressions or self-contained subqueries. The value must not be negative and the maximum number allowed is 9,223,372,036,854,775,807.

Notes

- Both the clauses are optional. However, if present, the OFFSET clause must come before the FETCH clause.
- If the OFFSET clause is specified without the FETCH clause, the SQL engine skips the specified number of rows and returns all other qualifying rows without an upper limit.
- If the FETCH clause is specified without the OFFSET clause, the SQL engine does not skip any rows and the initial value of the OFFSET is 0. This is similar to the TOP clause.
- The OFFSET and FETCH clauses need not be preceded by the ORDER BY clause. it is a good idea to use an ORDER BY clause that constrains the result rows into a unique order. If the ORDER BY clause is specified, the OFFSET and FETCH clauses must appear right after the ORDER BY clause of a query.
- The TOP clause cannot be combined with the OFFSET and FETCH clauses in the same query expression (in the same query scope).
- The TOP and OFFSET/FETCH clauses are mutually exclusive.
- The OFFSET and FETCH clauses cannot be used in the following instances:
  - As part of a subquery
  - Within the derived tables
  - Within the CREATE TABLE, CREATE VIEW, UPDATE, DELETE,and INSERT statements.
  - In queries used with set operators such as UNION, INTERSECT, and MINUS
  - In views

- The OFFSET and FETCH clauses are interpreted as being applied last, after all the other clauses specified in the query are applied.

Example

OFFSET clause in the SELECT statement

The following example demonstrates the OFFSET clause in the SELECT statement:

```
SELECT OrderID,OrderDate,custID,filler
FROM dbo.Orders OFFSET 10;
```

The above query skips the first 10 rows and returns the rest of the qualified rows.

FETCH clause in the SELECT statement

The following example demonstrates the FETCH clause in the SELECT statement:

```
SELECT OrderID,OrderDate,custID,filler
FROM dbo.Orders
ORDER BY OrderDate DESC, OrderID DESC
FETCH FIRST 10 ROWS ONLY;
```

The above query returns the first 10 rows without skipping any

OFFSET and FETCH clause in the SELECT statement

The following example demonstrates both the OFFSET and FETCH clauses in the SELECT statement:

```
SELECT OrderID,OrderDate,custID,filler
FROM dbo.Orders
ORDER BY OrderDate DESC, OrderID DESC
OFFSET 50 ROWS FETCH NEXT 10 ROWS ONLY;
```

The above query returns rows 51 through 60 in the result set of the query.

In the following example, the SELECT statement finds find employee details whose salary is less than 5th highest salary:

```
SELECT * FROM employee WHERE employee.salary <
 ( SELECT salary from salaries ORDER BY salary desc
OFFSET 5 rows FETCH NEXT 1 ROWS  only);
```

WITH clause

Enables table-level locking when a finer control of the types of locks acquired on an object is required. These locking hints override the current transaction isolation level for the session.

The locking hint clause, such as for READPAST, can only be specified in the main SELECT statement, but not in the subquery SELECT statement in the "search condition" of the WHERE clause.

Syntax

```
[ WITH ( READPAST NOLOCK [ WAIT timeout | NOWAIT ]) ]
```

Parameters

search_condition

The READPAST locking hint skips locked rows. This option causes a transaction to skip rows locked by other transactions that would ordinarily appear in the result set, rather than block the transaction waiting for the other transactions to release their locks on these rows. The READPAST lock hint applies only to transactions operating at READ COMMITTED isolation and will read only past row-level locks. Applies only to the SELECT statement.

The NOLOCK locking hint ensures records are not locked during the execution of a SELECT statement when the transaction isolation level is set to READ COMMITTED. When NOLOCK is invoked, a dirty read is possible. This locking hint only works with the SELECT statement.

WAIT timeout

Override the default lock-wait time out. The timeout value is in seconds and can be 0 or any positive number.

NOWAIT

Causes the SELECT statement to skip (read past) the row immediately if a lock cannot be acquired on a row in the selection set because of the lock held by some other transaction. The default behavior is for the transaction to wait until it obtains the required lock or until it times out waiting for the lock.

Example

The following example demonstrates the WITH clause in the SELECT statement:

```
SELECT * FROM Customer WHERE "CustNum" < 100 ORDER BY "CustNum" FOR
UPDATE
WITH (READPAST WAIT 1);
```

FOR UPDATE clause

Specifies update intention on the rows selected by the SELECT statement.

Syntax

```
FOR UPDATE [ OF [table.]column_name , ...][ NOWAIT ]
```

Parameters

OF [*table.*]*column_name* , ...

Specifies the table's column name to be updated.

NOWAIT

Causes the SELECT statement to return immediately with an error if a lock cannot be acquired on a row in the selection set because of the lock held by some other transaction. The default behavior is for the transaction to wait until it obtains the required lock or until it times out waiting for the lock.

Notes

If you specify FOR UPDATE, the database acquires exclusive locks on all the rows satisfying the SELECT statement. The database does not acquire row level locks if there is an exclusive lock on the table. See the LOCK TABLE statement for information on table locking.

Authorization

To select a specified column, you must have DBA privileges or SELECT permission on all the tables or views referred to in the *query_expression*.

A user mapped to a super-tenant can view data for all the tenants only if the user has the required permissions on the tables referenced in the statement.

Related Statements

WHERE clause, GROUP BY clause, HAVING clause, ORDER BY clause, FOR UPDATE clause.

SET CATALOG

Changes the default catalog name to be used for schema, table, and column references. The default catalog name is initially the name of the primary database.

Syntax

```
SET CATALOG catalog_name;
```

Parameter

catalog_name

Catalog name to be used as an alias for the database in schema, table and column references. This must be in the form of an SQL identifier of up to 32 bytes in length.

Notes

- The SET CATALOG statement is used to specify the default database catalog name to be used for schema, table, and column references.

- The primary database connection is automatically given a catalog name which is the name of the primary database. For example, if the primary database is at `/usr/progress/sports2020`, then the catalog name for the primary database is `sports2020`.
- The `SET CATALOG` statement may be used to set the default catalog to an auxiliary database catalog or to the primary database catalog.
- The specified catalog must identify a current catalog name.
- If an auxiliary database catalog is set as the default catalog, disconnecting from the auxiliary catalog will not change the name of the default catalog. Thus setting the default catalog to an auxiliary database may cause failures of statements when the default catalog is not set to an active catalog. In other words, shutting down an auxiliary database identified as the default catalog will cause any query using a three-part specification to fail. The failure produces an error indicating that the catalog is not connected.

Example

In this example, the auxiliary database connection identified by the catalog named `mydb1` is specified as the default catalog:

```
SET CATALOG mydb1;
```

Authorization

Any user is allowed to execute this statement.

SQL Compliance

Progress Software Corporation specific extension.

Related statements

CONNECT AS CATALOG, DISCONNECT CATALOG

SET DATABASE DEFAULT AREA

Sets the default area for tables, indexes, and LOBs.

Syntax

```
SET DATABASE DEFAULT AREA area_name FOR [ ALL|TABLE|INDEX|LOB ];
```

Parameters

- **`SET DATABASE DEFAULT AREA`**—Sets the default area for an object.
- **`area-name`**—Specifies the name of the storage area for which you want to create a default area.
- **`ALL`**—Specifies that all objects (tables, indexes, and LOBs) are created in the default area.

Note: You can use the same command to change the default areas. This command replaces the existing default area for the objects with a new default area.

SET PRO_CONNECT LOG

Controls logging for the current SQL Server connection.

Syntax

```
SET PRO_CONNECT LOG [ ON | OFF ][ WITH ({ERROR ONLY, STATEMENT, INOUT, QUERY_PLAN,
PRO_TABLESCAN_EVENT, ALL })]

[[AND] WITH LEVEL 1|LEVEL 2|LEVEL 3|LEVEL 4];
```

Parameters

`ON`

Indicates that logging is turned on.

`OFF`

Indicates that logging is turned off.

`ERROR ONLY`

Indicates that error tracing information is written to each log file.

`STATEMENT`

Indicates that statement tracing information is written to each log file.

`QUERY_PLAN`

Indicates that query plan information is written to the log file.

`INOUT`

Logs information to the output server log about the following:

- Input LOB data and LOB locator information
- Output LOB data
- Output parameters for statements like the select list for the SELECT statement

`PRO_TABLESCAN_EVENT`

Information about tables on which table scan took place in the query.

`ALL`

It enables logging behavior for all other supported parameters such as STATEMENT, INOUT, QUERY_PLAN, and PRO_TABLESCAN_EVENT.

`Log Level`

The level of detail in the log entry depends on the level of logging that is chosen. Level 1 logs the most basic and most important information. Level 2 adds more details, and Level 3 adds still more details. Level 4 adds detailed information about connection events, when the client connects to SQL and to the database.

Notes

- When logging is set `ON`, the current SQL connection begins logging to a file named as `SQL_server_<`*`server-id`*`>_<`*`thread-id`*`>_<`*`yyyymmdd`*`>_<`*`hhmmss`*`>_<`*`A/B`*`>.log.`

  For example: `SQL_connection_1_14643_20180425_043722_A.log`

- The *`server-id`* corresponds to the server ID shown in *`database_name`*`.log.`
- Logging files are created in the location from where you execute the proserve command.
- The maximum size of each logging file is 500 MB. When `SQL_server_<`*`server-id`*`>_<`*`thread-id`*`>_<`*`yyyymmdd`*`>_<_<`*`hhmmss`*`>_<`*`A`*`>.log` reaches 500 MB, the OE SQL Server logs a message indicating the file was closed due to reaching the maximum size. After this message is written, the OE SQL server ceates another file with suffix '_B' and writes into it. For every recurrence of 500 MB limit, the file contents are overwritten in a round-robin way between these two files.
- When logging commences to a new file, the file contents begin with information about the SQL Server environment, including:
  - Environment variable settings
  - Logging the options passed while enabling the server log
  - Logging control values (such as size limits)
  - The SQL Server process ID
- Each section of information written to the log file begins with the string
- *`YYYYMMMDD_HH:MM:SS`* < *`user-id`* >:

  For example, 120180425_050934 37825:

SET PRO_CONNECT QUERY_TIMEOUT

Defines the maximum number of seconds during which a query should execute for the current SQL Server connection.

Syntax

```
SET PRO_CONNECT QUERY_TIMEOUT n ;
```

Parameter

n

Indicates the maximum number of seconds during which a query should execute before it is automatically cancelled by the SQL server. Setting the value of *n* to `0` disables a previously set query timeout.

Notes

- Both database and non-database administrators can execute this command. The value specified by *n* applies only to current SQL connection and is in effect for the duration of the current connection.
- Timeout value can be cleared by specifying a value of 0 on subsequent execution of the statement `SET PRO_CONNECT QUERY_TIMEOUT 0`.
- To enable timeout at server level use the `PRO_SERVER` statement, `SET PRO_SERVER QUERY_TIMEOUT n`
- When both the Server and connection timeouts are set, the lower of the timeout values takes the precedence.
- To set timeout when working with JDBC and ODBC clients, execute the `SET PRO_SERVER/CONNECT` statement before executing the SQL statements.
- Timeout is applicable to only SELECT statements. Other DDL and DML statements are not cancelled by SQL server when they exceed the specified timeout value.
- Timeout accounts only the query execution time i.e., the time it takes to prepare, execute and fetch the results. It does not account the time the clients take to stream the data.

Example

This example sets the query timeout to `30 seconds`:

```
Statement stmt = connection.createStatement();
String MySetQueryTimeout;
String MyClearQueryTimeout;
MySetTimeout = "SET PRO_CONNECT QUERY_TIMEOUT 30";
MyClearTimeout = "SET PRO_CONNECT QUERY_TIMEOUT 0";
// Set SQL Server timeout for query execute and fetch
stmt.executeUpdate( MySetQueryTimeout );
// Add code here to perform queries
// Clear SQL Server timeout for query execute and fetch
stmt.executeUpdate( MyClearQueryTimeout );
```

SET PRO_SERVER QUERY_TIMEOUT

Defines the maximum number of seconds during which a query should execute for all the client connections spawned by the SQL Server.

Syntax

```
SET PRO_SERVER QUERY_TIMEOUT n ;
```

Parameters

n

Indicates the maximum number of seconds during which a query should execute before it is automatically cancelled by the SQL server. Setting the value of *n* to `0` disables a previously set query timeout.

Notes

- Execution of this command is restricted to database administrators. Any value set with this command is in effect for the duration that the database is up and running.
- The value specified by n applies to all connections of the SQL server until the timeout value is cleared.
- Timeout value can be cleared by specifying a value of 0 on subsequent execution of the statement `SET PRO_SERVER QUERY_TIMEOUT 0`.
- To enable timout for an individual connection, use PRO_CONNECT statement `SET PRO_CONNECT QUERY_TIMEOUT N`.
- When both the server and connection timeouts are set, the lower of the timeout values takes the precedence.
- To set timeout when working with JDBC and ODBC clients, execute the `SET PRO_SERVER/CONNECT` statement before executing the SQL statements.
- Timeout is applicable to only select statements. Other DDL and DML statements are not cancelled by SQL server when they exceed the specified timeout value.
- Timeout accounts only the query execution time i.e., the time it takes to prepare, execute and fetch the results. It does not account the time the clients take to stream the data.

This example sets the query timeout to `30 seconds`:

```
Statement stmt = connection.createStatement();
String MySetQueryTimeout;
String MyClearQueryTimeout;
MySetTimeout = "SET PRO_SERVER QUERY_TIMEOUT 30";
MyClearTimeout = "SET PRO_SERVER QUERY_TIMEOUT 0";
// Set SQL Server timeout for query execute and fetch
stmt.executeUpdate( MySetQueryTimeout );
// Add code here to perform queries
// Clear SQL Server timeout for query execute and fetch
stmt.executeUpdate( MyClearQueryTimeout );
```

SET PRO_SERVER LOG

Controls logging for all connections to all OpenEdge SQL Servers.

Syntax

```
SET PRO_SERVER LOG [ ON | OFF ]
[ WITH ({ERROR_ONLY, STATEMENT, INOUT, QUERY_PLAN, PRO_TABLESCAN_EVENT, ALL})]
[[AND] WITH LEVEL 1|LEVEL 2|LEVEL 3|LEVEL 4];
```

Parameters

`ON`

Indicates that logging is turned on.

`OFF`

Indicates that logging is turned off.

ERROR ONLY

Indicates that error tracing information is written to each log file.

STATEMENT

Indicates that statement tracing information is written to each log file.

QUERY_PLAN

Indicates that query plan information is written to the log file.

INOUT

Logs information to the output server log about the following:

- Input LOB data and LOB locator information
- Output LOB data
- Output parameters for statements like the select list for the SELECT statement

PRO_TABLESCAN_EVENT

Information about tables on which table scan took place in the query.

ALL

It enables logging behavior for all other supported parameters such as STATEMENT, INOUT, QUERY_PLAN, and PRO_TABLESCAN_EVENT.

Log Level

The level of detail in the log entry depends on the level of logging that is chosen. Level 1 logs the most basic and most important information. Level 2 adds more details, and Level 3 adds still more details. Level 4 adds detailed information about connection events, when the client connects to SQL and to the database.

Notes

- When logging is set ON, each SQL Server begins logging to a file named as SQL_server_<*server-id*>_<*yyyymmdd*>_<*hhmmss*>_<*A/B*>.log

  For example: SQL_server_1_20180425_043429_A.log

- The *server-id* corresponds to the server ID shown in *database_name*.lg.
- Logging files are created in the location from where you execute the proserve command.
- The maximum size of each logging file is 500 MB. When SQL_server_<*server-id*>_<*yyyymmdd*>_<*hhmmss*>_<*A*>.log reaches 500 MB, the OE SQL Server logs a message indicating the file was closed due to reaching the maximum size. After this message is written, the OE SQL server ceates another file with suffix '_B' and writes into it. For every recurrence of 500 MB limit, the file contents are overwritten in a round-robin way between these two files.
- When logging commences to a new file, the file contents begin with information about the SQL Server environment, including:
  - Environment variable settings
  - Logging the options passed while enabling the server log

- Logging control values (such as size limits)
- The SQL Server process ID

- Each section of information written to the log file begins with the string
- *YYYYMMMDD_HH:MM:SS* < *user-id* >:

  For example, 120180425_050934 37825:

SET ROWCOUNT

Limits the result set to a specified number of rows. This functionality of restricting the size of the result set is similar to that of the TOP clause. However, the TOP clause only applies to queries at the statement level, while SET ROWCOUNT is a session-wide parameter and is not statement-specific.

Syntax

```
SET ROWCOUNT rowcount_number;
```

Parameter

rowcount_number

Specifies the number of rows to be restricted from the result set. This value must be a constant literal.

Notes

- The default value of the row countis 0, which means that the number of rows in the result set is not restricted, as shown in the example above. When the row count value is greater than 0, the maximum size of the result set is that of the row count value.
- You can change the row count value at any point of time in a session. The new value affects the subsequent SELECT statements and stored procedures in the session.
- SET ROWCOUNT restricts the result set after all the statement level clauses (such as, OFFSET, FETCH, and TOP clauses) have been applied. This implies that after the restriction of OFFSET, FETCH, and TOP clauses on the result set, SET ROWCOUNT further restricts the result set.
- The SET ROWCOUNT statement has no effect on the following:
  - DDL statements such as CREATE, ALTER, and DROP
  - DML statements such as INSERT, DELETE, and UPDATE
  - SHOW statements
  - Triggers
- The SET ROWCOUNT statement has no effect on the internal SELECT statements used in stored procedures. However, the overall result set of the stored procedures is restricted by SET ROWCOUNT, as shown in the example above.
- The SET ROWCOUNT statement affects the overall query result and not the intermediate stages of a query execution as the following:

- Intermediate result set in a subquery
- Intermediate result set in views or derived tables
- Intermediate queries used as set operators such as, UNION, INTERSECT, and MINUS

Examples

SET ROWCOUNT statement

The following examples demonstrate the SET ROWCOUNT statement:

```
SET ROWCOUNT 10;
SELECT * FROM Customer;
```

The above query returns the first ten rows in the result set.

SET ROWCOUNT statement with OFFSET and FETCH clauses

The following example demonstrates the SET ROWCOUNT statement with OFFSET and FETCH clauses in the SELECT statement:

```
SET ROWCOUNT 5;
SELECT * FROM Customer
ORDER BY Name
OFFSET 10 ROWS FETCH NEXT 10 ROWS ONLY;
```

The above query further restricts the result set, after applying the OFFSET and FETCH clauses, and returns rows 11 through 15.

SET ROWCOUNT statement

The following example demonstrates the SET ROWCOUNT statement, where the row count value is set to 3:

```
SET ROWCOUNT 3;
SELECT TOP 5 * FROM Customer;
```

The above query returns only the first 3 rows in the result set.

Consider running the same query with the row count value set to 0 (the default value for SET ROWCOUNT), as shown below:

```
SET ROWCOUNT 0;
SELECT TOP 5 * FROM Customer;
```

The above query returns the first 5 rows in the result set without restricting any rows.

SET ROWCOUNT statement in a stored procedure

The following example demonstrates the SET ROWCOUNT statement in a stored procedure containing more than one row:

```
SET ROWCOUNT 1;
CALL get_sal ();
```

The above query returns only the first row in the result set.

Related Statement

SHOW ROWCOUNT

SET SCHEMA

Sets the default owner, also known as schema, for unqualified table references.

Syntax

```
SET SCHEMA { 'string_literal' | ? | USER }
```

Parameters

'string_literal'

Specifies the name for the default owner as a string literal, enclosed in single or double quotes.

Indicates a parameter marker to contain the default owner. The actual replacement value for the owner name is supplied in a subsequent SQL operation.

USER

Directs the database to set the default owner back to the *username* that established the session.

Notes

- For authorization purposes, invoking SET SCHEMA does not change the *username* associated with the current session.
- You can set the default schema name to the *username* associated with the session by using a SET SCHEMA USER statement.

Example

This example sets the default schema name to White:

```
SET SCHEMA 'White' ;
COMMIT ;
 SELECT * from customer ;
```

Subsequent SQL statements with unqualified table references will use the owner name `White`. The `SELECT` statement in this example returns all rows in the 'White.customer' table. The *`username`* establishing the original session is still the current user.

Authorization

None

SET TRANSACTION ISOLATION LEVEL

Explicitly sets the isolation level for a transaction. Isolation levels specify the degree to which one transaction can modify data or database objects in use by another concurrent transaction.

Syntax

```
SET TRANSACTION ISOLATION LEVEL isolation_level_name ;
```

Parameters

`isolation_level_name`

The following is the syntax for *`isolation_level_name`*:

```
READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE
```

`READ UNCOMMITTED`

Also known as a dirty read. When this isolation level is used, a transaction can read uncommitted data that later might be rolled back. The standard requires that a transaction that uses this isolation level can only fetch data but cannot update, delete, or insert data.

`READ COMMITTED`

Dirty reads are not possible with this isolation level. However, if the same row is repeatedly read during the same transaction, its contents can be changed or the entire row can be deleted by other transactions.

`REPEATABLE READ`

This isolation level guarantees that a transaction can read the same row many times and it will remain intact. However, if a query with the same search criteria (the same `WHERE` clause) is executed more than once, each execution can return different sets of rows. This can happen because other transactions are allowed to insert new rows that satisfy the search criteria or update some rows in such a way that they now satisfy the search criteria.

`SERIALIZABLE`

This isolation level guarantees that none of the above happens. Transactions that use this level will be completely isolated from other transactions.

Notes

- See the LOCK TABLE for information on record locking schemes used by each isolation level.
- For more information on transactions, see *OpenEdge Data Management: SQL Development*.

Authorization

None

Related statements

COMMIT, LOCK TABLE, ROLLBACK

SHOW CATALOGS

Returns a list of available catalog information with catalog name, catalog type (primary or auxiliary), and catalog status (default or not default).

Syntax

```
SHOW CATALOGS [ ALL |{ PRO_NAME | PRO_TYPE | PRO_STATUS }
              [ , PRO_NAME | PRO_TYPE | PRO_STATUS } ;
```

Parameters

`ALL`

Return all attributes.

`PRO_NAME`

List of catalog names.

`PRO_TYPE`

List of catalog types (primary or auxiliary).

`PRO_STATUS`

List of catalog statuses (default or notdefault)

Notes

- This statement is useful for obtaining the catalog names of databases currently connected for the user, for identifying the catalog name of the primary database (automatically connected) and the current default catalog.
- The primary database connection is automatically given a catalog name which is the name of the primary database. For example, if the database is at `/usr/progress/sports2000`, then the catalog name for the primary database is `sports2000`.

- Three columns of information are returned by the SHOW CATALOG statement. These are the catalog name, catalog type (primary or auxiliary) and status (default or notdefault).

Example

In this example, the names of the currently available catalogs are returned.

```
SHOW CATALOGS PRO_NAME;
```

Authorization

Any user is allowed to execute this statement.

SQL Compliance

Progress Software Corporation specific extension.

Related statements

CONNECT AS CATALOG, DISCONNECT CATALOG, SET CATALOG

SHOW DATABASE DEFAULT AREA

Displays the default areas for table, index, and LOBs.

Syntax

```
SHOW DATABASE DEFAULT AREA [ ALL | TABLE | INDEX | LOB ];
```

Parameters

- **SHOW DATABASE DEFAULT AREA**—Displays the default area for an object.
- **ALL | TABLE | INDEX | LOB** —The object for which you want to display the default area. You can view the default areas for all objects by using ALL or for specific objects by using TABLE, INDEX, or LOB

Note: If you did not set the default area for any object type, then this command returns an empty result.

SHOW GROUP

Returns a result set with information about the database tables and the existing tenants that are associated with one or more groups.

Syntax

```
SHOW { ALL GROUP [{ FOR TABLE table_name | FOR TENANT tenant_name }]| GROUP FOR
 group_name}
      [ FULL | PRO_NAME ;]
```

Parameters

SHOW ALL GROUP

Returns information about all the groups defined in the multi-tenant database.

FOR TABLE *table_name*

Returns information about all the groups defined for the specified table. Returns zero rows if the specified table is not a multi-tenant table.

FOR TENANT *tenant_name*

Returns information about all the groups that the tenant belongs to.

SHOW GROUP FOR *group_name*

Returns information about the specified group.

FULL

Returns all the information about all the groups defined for the specified table.

PRO_NAME

Returns only the names of the groups defined for the specified table.

Example

The following example shows how to return the basic information about all the groups defined in the database:

```
SHOW ALL GROUP;
```

The following example shows how to return all the group names defined for a table:

```
SHOW ALL GROUP FOR TABLE pub.mtcustomer PRO_NAME;
```

The following example shows how to get full information about all the groups defined for a table:

```
SHOW ALL GROUP FOR TABLE pub.mtcustomer FULL;
```

The following example shows how to get information about all the groups that a tenant belongs to:

```
SHOW ALL GROUP FOR TENANT Consolidated_Freightways;
```

Authorization

Execution of the SHOW GROUP statement requires DBA privileges.

Related Statement

CREATE GROUP

SHOW PARTITION

Allows a security administrator or a DBA to display partition information (meta data) about the primary database. It returns a result set that is ordered by a particular column based on the type of statement. The columns are determined by the options specified.

The SHOW PARTITION FULL statement also specifies if a partition is a read-only partition or a read-write partition. The READ_ONLY column with YES as its value indicates that the partition is marked as a read-only partition. With NO as its value, the READ_ONLY column indicates that the partition is marked as a read-write partition.

The SHOW PARTITION statement uses the following syntax:

Syntax

```
SHOW { ALL PARTITION | PARTITION FOR partition_name }
  [table_option]
  [show_options]
  [ COLUMN SIZE size]
```

Parameters

table_option

Uses the following syntax:

```
FOR TABLE table_name
```

FOR TABLE *table_name*

Returns information about all the partitions defined for the specified table.

show_options

Uses the following syntax:

```
FULL | PRO_NAME | TABLE | PRO_AREA_NAME
```

FULL

Returns all the information about all the partitions defined for the specified table.

PRO_NAME

Returns only the names of the partitions defined for the specified table.

PRO_AREA_NAME

Returns information about the names of the storage areas used for each partition. It does not return any information about the partition key columns.

Examples

The following example returns the basic information about all the partitions in the database:

```
SHOW ALL PARTITION
```

The following example returns all the partition names defined in the database:

```
SHOW ALL PARTITION PRO_NAME;
```

The following example returns all the information about all partitions defined in the database:

```
SHOW ALL PARTITION FULL;
```

The following example returns the basic information about all the partitions defined on the table `Pub.Customer`:

```
SHOW ALL PARTITION FOR TABLE Pub.Customer;
```

The following example returns all the information about all partitions created on the table `Pub.Customer`. It also limits the column values in the result set to not more than 30 characters.

```
SHOW ALL PARTITION FOR TABLE Pub.Customer FULL COLUMN SIZE 3;
```

The following example returns a list of partitioned tables:

```
SHOW ALL PARTITION TABLE
```

The following example returns information about partitions for table `Pub.tpcustomer` including names of the storage areas containing data, indexes, and LOB columns:

```
SHOW ALL PARTITION FOR TABLE Pub.tpcustomer PRO_AREA_NAME
```

Authorization

Execution of the `SHOW PARTITION` statement requires DBA privileges.

SHOW ROWCOUNT

Displays the current value of the row count.

Syntax

```
SET ROWCOUNT rowcount_number;SHOW ROWCOUNT;
```

Example

The following example demonstrates the `SHOW ROWCOUNT` statement:

```
SET ROWCOUNT 10;
SHOW ROWCOUNT;
```

The above query returns the row count value as 10.

Related Statement

SET ROWCOUNT

SHOW TENANT

Returns specific information for a tenant. The information includes type and status of the tenant. If the tenant type is default, the `SHOW TENANT` statement displays the tenant ID as zero.

Syntax

```
SHOW [ ALL TENANT | TENANT FOR tenant_name ][ FOR TABLE table_name ]
  [ FULL | PRO_NAME | TABLE | TABLE INDEX ];
```

Parameters

SHOW ALL TENANT

Returns information about the database objects for the defined tenants.

SHOW TENANT FOR tenant_name

Returns information about the database objects for the specified tenant.

FOR TABLE table_name

Returns information about the tenant in the specified table. If the specified table is not a multi-tenant table, it does not return any rows in the result set.

FULL

Returns complete information about the database objects for the specified tenant.

PRO_NAME

Returns the names of tenants in the multi-tenant table.

TABLE

Returns information about the table associated with the specified tenant.

TABLE INDEX

Returns information about the table and index defined for the multi-tenant tables associated with a specified tenant.

Example

The example shows how to return the available tenants of a table mtcustomer:

```
SHOW ALL TENANT FOR TABLE pub.mtcustomer;
```

Authorization

All users are allowed to execute the SHOW TENANT statement.

Related Statement

DROP TENANT

SHOW ENCRYPT ON

The SHOW ENCRYPT statement provides encryption policy information on the primary database. It can be used only by security administrators or DBAs.

Syntax

```
SHOW ENCRYPT ON { ALL | [ TABLE | INDEX | LOB ]
                      | TABLE table_name [ WITH INDEX | WITH LOB ]
                      | TABLE table_name ON INDEX index_name } ;
```

When run, the statement returns a result set with eight columns:

- Database object type (AREA, TABLE, INDEX, LOB)
- Database object name
- Object's table name (blank for area)
- Database object name (blank for area)
- Database object identification
- Object policy state (CURRENT or PREVIOUS)
- Object policy cipher name
- Object policy version number

Notes

- Only active policies are returned by the statement.
- The only option which shows Type I area encryption information is the SQL statement SHOW ENCRYPTION ON ALL. Other options on SHOW ENCRYPT show encryption information only for Type II area database objects.

Authorization

Security Administrator or DBA.

SQL Compliance

Progress Software Corporation specific extension.

Related statements

ALTER TABLE, CREATE INDEX, CREATE TABLE

Table constraints

Specifies a constraint for a table that restricts the values that the table can store. INSERT, UPDATE, or DELETE statements that violate the constraint fail. SQL returns a constraint violation error.

Table constraints have syntax and behavior similar to column constraints. Note the following differences:

- The definitions of the table constraints are separated from the column definitions by commas.
- Table constraint definitions can include more than one column, and SQL evaluates the constraint based on the combination of values stored in all the columns.

Syntax

```
CONSTRAINT constraint_name
    PRIMARY KEY ( column [ , ... ] )
  | UNIQUE ( column [ , ... ] )
  | FOREIGN KEY ( column [, ... ] )
     REFERENCES [owner_name.]table_name [ ( column [ , ... ] ) ]
    | CHECK ( search_condition )
```

Parameters

CONSTRAINT *constraint_name*

Allows you to assign a name that you choose to the table constraint. While this specification is optional, this facilitates making changes to the table definition, since the name you specify is in your source CREATE TABLE statement. If you do not specify a *constraint_name*, the database assigns a name. These names can be long and unwieldy, and you must query system tables to determine the name.

PRIMARY KEY (*column* [, ...])

Defines the column list as the primary key for the table. There can be at most one primary key for a table.

All the columns that make up a table level primary key must be defined as NOT NULL, or the CREATE TABLE statement fails. The combination of values in the columns that make up the primary key must be unique for each row in the table.

Other tables can name primary keys in their REFERENCES clauses. If they do, SQL restricts operations on the table containing the primary key in the following ways:

- DROP TABLE statements that delete the table fail
- DELETE and UPDATE statements that modify values in the combination of columns that match a foreign key's value also fail

UNIQUE (*column* [, ...])

Defines the column list as a unique, or candidate, key for the table. Unique key table-level constraints have the same rules as primary key table-level constraints, except that you can specify more than one UNIQUE table-level constraint in a table definition.

FOREIGN KEY (*column* [, ...]) REFERENCES [*owner_name*.]*table_name* [(*column* [, ...])]

Defines the first column list as a foreign key and, in the REFERENCES clause, specifies a matching primary or unique key in another table.

A foreign key and its matching primary or unique key specify a referential constraint. The combination of values stored in the columns that make up a foreign key must either:

- Have at least one of the column values be null.
- Be equal to some corresponding combination of values in the matching unique or primary key.

You can omit the column list in the REFERENCES clause if the table specified in the REFERENCES clause has a primary key and you want the primary key to be the matching key for the constraint.

CHECK (*search_condition*)

Specifies a table level check constraint. The syntax for table level and column level check constraints is identical. Table level check constraints must be separated by commas from surrounding column definitions.

SQL restricts the form of the search condition. The search condition must not:

- Refer to any column other than columns that precede it in the table definition
- Contain aggregate functions, subqueries, or parameter references

Examples

In the following example, which shows creation of a table level primary key, note that its definition is separated from the column definitions by a comma:

```
CREATE TABLE SupplierItem (
     SuppNum   INTEGER NOT NULL,
     ItemNum   INTEGER NOT NULL,
     Quantity       INTEGER NOT NULL DEFAULT 0,
          PRIMARY KEY (SuppNum, ItemNum)) ;
```

The following example shows how to create a table with two UNIQUE table level constraints:

```
CREATE TABLE OrderItem (
     OrderNum    INTEGER NOT NULL,
     ItemNum     INTEGER NOT NULL,
     Quantity         INTEGER NOT NULL,
     Price       INTEGER NOT NULL,
          UNIQUE (OrderNum, ItemNum),
          UNIQUE (Quantity, Price));
```

The following example defines the combination of columns `student_courses.teacher` and `student_courses.course_title` as a foreign key that references the primary key of the `courses` table:

```
CREATE TABLE Courses (
     Instructor        CHAR (20) NOT NULL,
     CourseTitle   CHAR (30) NOT NULL,
     PRIMARY KEY (Instructor, CourseTitle));
CREATE TABLE StudentCourses (
     StudentID     INTEGER,
     Instructor        CHAR (20),
     CourseTitle   CHAR (30),
     FOREIGN KEY (Instructor, CourseTitle) REFERENCES Courses);
```

Note that this `REFERENCES` clause does not specify column names because the foreign key refers to the primary key of the `courses` table.

SQL evaluates the referential constraint to see if it satisfies the following search condition:

```
(StudentCourses.Ieacher IS NULL
     OR StudentCourses.CourseTitle IS NULL)
     OR EXISTS (SELECT * FROM StudentCourses WHERE
          (StudentCourses.Instructor = Courses.Instructor AND
          StudentCourses.CourseTitle = Courses.CourseTitle));
```

Note: `INSERT`, `UPDATE`, or `DELETE` statements that cause the search condition to be false violate the constraint, fail, and generate an error.

In the following example, which creates a table with two column level check constraints and one table level check constraint, each constraint is defined with a name:

```
CREATE TABLE supplier (
     SuppNum   INTEGER NOT NULL,
     Name      CHAR (30),
     Status    SMALLINT CONSTRAINT StatusCheckCon
                   CHECK (Supplier.Status BETWEEN 1 AND 100 ),
     City      CHAR (20) CONSTRAINT CityCheckCon CHECK
                    (Supplier.City IN ('New York', 'Boston', 'Chicago')),
     CONSTRAINT SuppTabCheckCon CHECK (Supplier.City <> 'Chicago'
          OR Supplier.Status = 20)) ;
```

UPDATE

Updates the rows and columns of the specified table with the given values for rows that satisfy the *`search_condition`*.

When updating row(s) of a multi-tenant table, a regular tenant can only update rows in its partition, and the rows remain in the same tenant partition, but a super-tenant or a DBA can update rows in all the tenant partitions and group partitions. A super-tenant or a DBA may refine which tenants' rows must be affected by using the `tenantid_tbl()` or the `tenantName_tbl()` function in the `WHERE` clause `search_condition`.

Updating a row of a partitioned table may result in one of the following:

- If unpartitioned key columns are updated, then the updated new record remains in its original partition. In this case, there is no change in the behavior of the `UPDATE` statement.

- If only one RANGE partition key column is updated and the updated value is in the existing partition range, then the record remains in the existing partition.
- If one or more LIST partition key columns are being updated (or the RANGE partition key column is updated and the new value are not in the existing partition), then the updated new record is moved to a different partition. The new partition ID is determined with the newly updated record's partition key columns.
- If the new record's partition key columns cannot determine any partition while updating the record, then the UPDATE statement returns an error stating that the updated row does not belong to any partition.
- If the row being updated belongs to a read-only partition, then the UPDATE statement returns an error. The UPDATE statement also returns an error if the row being updated belongs to a read-write partition and the target partition is a read-only partition.

Note: You must have the write permissions on the partitioned table to update its records.

Syntax

```
UPDATE table_name
  SET assignment [, assignment], ...
     [ WHERE search_condition] ;
```

Parameter

assignment:

The syntax for *assignment* is given below:

```
column = {expr | NULL }
   | (column [, column], ... ) = (expr [, expr])
   | (column[, column] , ... ) = (query_expression)
```

Notes

- If you specify the optional WHERE clause, only rows that satisfy the *search_condition* are updated. If you do not specify a WHERE clause, all rows of the table are updated.
- If the expressions in the SET clause are dependent on the columns of the target table, the expressions are evaluated for each row of the table.
- If a query _expression is specified on the right-hand side of an assignment, the number of expressions in the first SELECT clause must be the same as the number of columns listed on the left-hand side of the assignment.
- If a query _expression is specified on the right-hand side of an assignment, it must return a single row.
- If a table has check constraints and if the columns to be updated are part of a check expression, then the check expression is evaluated. If the result of the evaluation is FALSE, then, the UPDATE statement fails.
- If a table has a primary or candidate key, and if the columns to be updated are a part of the primary or candidate key, SQL checks to determine if there is a corresponding row in the referencing table. If there is no corresponding row, then, the UPDATE statement fails.

Column names in the SET clause do not need a table_name qualifier. Since an UPDATE statement affects a single table, columns in the SET clause are implicitly qualified to the table name identified in the UPDATE clause.

Examples

The following examples illustrate the UPDATE statement:

The example below illustrates the different forms of UPDATE statement.

```
UPDATE Orderline
     SET Qty = 186
     Where Ordernum = 22;
  Update Orderline
      SET (Itemnum) =
                     (Select Itemnum
                      FROM Item
                      WHERE Itemname = 'Tennis balls')
     WHERE Ordernum = 20;

UPDATE Orderline
          SET (Qty) = (200 * 30)
          WHERE OrderNum = 19;
  UPDATE OrderLine
          SET (ItemNum, Price) =
                              (SELECT ItemNum, Price * 3
                               FROM Item
                               WHERE ItemName = 'gloves')
          WHERE OrderNum = 21 ;
```

the example below updates the postal code to '99999' for a customer '1428' for the tenant SNCSoftware.

```
UPDATE pub.mtcustomer
SET postalcode = '99999'
WHERE custnum = 1428 AND tenantName_tbl (pub.mtcustomer) = 'SNCSoftware';
```

The example below updates the postal code to '99999' for the customer '1428' for all tenants:

```
UPDATE pub.mtcustomer
SET postalcode = '99999'
WHERE custnum = 1428;
```

Updating partitioned tables

The following examples illustrate updating rows of a table that is partitioned by RANGE.

Note: Updating a partition key column value may result in moving rows from one partition to another.

Assume that the table Pub.tporder is partitioned by RANGE based on the column OrderDate as given below:

- PART1_RANGE OrderDate <= ('01/01/1998')
- PART2_RANGE OrderDate <= ('01/01/2010')
- PART3_RANGE OrderDate <= ('01/01/2018')

Assume that the following rows exist in the table:

- (OrderNum, Custnum, OrderDate)
- (1, 100, '10/10/1990')
- (2, 101, '10/10/2017')

The below UPDATE statements do not change the partition of the row:

```
UPDATE Pub.tporder
SET OrderDate = '01/05/1997'
WHERE OrderNum = 1;
```

```
UPDATE Pub.tporder
SET Custnum = 103
WHERE OrderNum = 1;
```

The below UPDATE statement results in changing the partition of a row and moves the row from partition PART3_RANGE to PART1_RANGE:

```
UPDATE Pub.tporder
SET OrderDate = '01/05/1997'
WHERE OrderNum = 2;
```

The below UPDATE statement results in an error since the row does not belong to any partition:

```
UPDATE Pub.tporder
SET OrderDate = '01/01/2019'
WHERE OrderNum = 1;
```

Authorization

To update rows and columns of a particular table, you must have DBA privileges or UPDATE privileges on all the specified columns of the target table, and SELECT privilege on all the other tables referred to in the statement.

A user mapped to a super-tenant can view data for all the tenants only if the user has the required permissions on the tables referenced in the statement.

Related statements

SELECT, OPEN, FETCH

UPDATE STATISTICS

Queries data tables and updates the following statistics:

- Table cardinality
- Index statistics
- Column data distribution for columns that are index components

- Column data distribution for columns that are not index components

Syntax

```
UPDATE ([ TABLE | INDEX
 |[ ALL ] COLUMN ] STATISTICS [ AND ]) ... [ FOR table_name[ FOR FOR PARTITIONS
 partition_name_1, [partition_name_n, …]] [FOR TENANTS tenant1, [tenant2, …]]
 | FOR CDC SYSTEM TABLES] [WITH SAMPLE PERCENT n];
```

Notes

- In the Update Statistics syntax, partition_name_1,…,partition_name_n are partition names of partitions belonging to table table_name.
- FOR PARTITIONS clause can be used only with partitioned tables and it fetches statistics only from the specified partitions in the partition list.
- FOR PARTITIONS clause, on updating index statistics, updates index statistics from only local indexes for the specified partitions in partition list and does not update global index statistics.
- All statistics are obtained online. Obtaining statistics does not require an exclusive lock on the schema or any table locks. Rows written to statistics tables will be exclusively locked, as in every transaction performing updates. Therefore, statistics can be obtained while normal database operations continue.
- Specifying `TABLE STATISTICS` obtains table cardinality only. Table cardinalities are stored in the `SYSTABLSTAT` system catalog table.
- Specifying `INDEX STATISTICS` obtains statistics on the number of unique values in each index. Index statistics are stored in the `SYSIDXSTAT` system catalog table.
- Specifying `COLUMN STATISTICS` (without `ALL`) obtains statistics on the data distribution of values for each column that is an index key component.
- Specifying `ALL COLUMN STATISTICS` obtains statistics on the data distribution of values for all columns.
- The `STATISTICS` phrase can be repeated so that up to three statistics can be requested by a single `UPDATE STATISTICS` statement.
- By default, for the simple statement `UPDATE STATISTICS`, where the type of statistics is not specified, SQL will obtain table and index column statistics. This is equivalent to the statement `UPDATE TABLE STATISTICS AND COLUMN STATISTICS`.
- A table containing `LONG` data types can get table, index, and/or column statistics. The columns that are `LONG` data types cannot get statistics.
- Obtaining table statistics runs in time proportional to the table's primary index.
- Obtaining column statistics runs in time proportional to the table's primary index, plus an additional amount proportional to the number of columns in the table.
- The UPDATE STATISTICS command for data of a sample percentage is applicable only for column statistics. This is not applicable for index and table statistics.
- Obtaining index statistics runs in time proportional to the total size for all indexes for the table.
- Table statistics are often the most useful statistic, as they influence join order substantially.
- Index statistics are important when a table has five or more indexes. This is especially true if some of the indexes are similar to one another.

- Column statistics are the most useful when applications use range predicates, such as BETWEEN and the operators **<**, **<=**, **>** and **>=**.
- The FOR TENANTS and FOR PARTITIONS clauses are mutually exclusive.
- The FOR TENANTS clause can be used with multi-tenant tables to create statistics for only the specified tenants in the tenants list. When this caluse is not used, statistics are created for all tenants.
- The tenant specified in the FOR TENANTS clause should not be a super-tenant.
- The phrase FOR CDC SYSTEM TABLES is an alternative to the phrase FOR <tbl> and is the equivalent of running the UPDATE STATISTICS statement for all indivudual CDC system tables. The CDC system tables are as follows:
  - pub."_Cdc-Change-Tracking"
  - pub."_Cdc-Table-Policy"
  - pub."_Cdc-Field-Policy"
  - pub."_File"

Note:

To get the best SQL query performance, or if a SQL performance problem occurs, be sure that the database has a full set of SQL statistics. To get a full set of SQL statistics, execute this SQL statement:

```
UPDATE TABLE STATISTICS AND INDEX STATISTICS AND COLUMN STATISTICS;
```

At a slightly longer execution time, you can get even better SQL statistics by doing:

```
UPDATE TABLE STATISTICS AND INDEX STATISTICS AND ALL COLUMN STATISTICS;
```

Examples

The following example shows default commands for table cardinality and data distribution for index component columns:

```
UPDATE STATISTICS FOR Customer;
```

The following example shows commands for table cardinality only:

```
UPDATE TABLE STATISTICS FOR Customer;
```

The following example shows commands for new index statistics:

```
UPDATE INDEX STATISTICS FOR Customer;
```

The following example shows commands for updating column statistics for index columns only:

```
UPDATE COLUMN STATISTICS FOR Customer;
```

The following example shows commands for updating statistics for all columns:

```
UPDATE ALL COLUMN STATISTICS FOR Customer;
```

The following example shows commands for updating statistics for columns with a sample percentage:

```
UPDATE ALL COLUMN STATISTICS FOR Pub.Customer WITH SAMPLE PERCENT 5;
```

Note: Sample percentage must be a value between 0 to 100 and can be a decimal value. For example, 0.6.

The following example shows commands to obtain table cardinality and new index statistics and column statistics for all columns:

```
UPDATE TABLE STATISTICS AND INDEX STATISTICS AND ALL COLUMN STATISTICS FOR Customer;
```

The following example updates table, and column statistics of partitions `USA_Customer`, `EUROPE_Customer` and `ASIA_Customer` for table Customer:

```
UPDATE STATISTICS FOR Customer FOR PARTITIONS USA_Customer, EUROPE_Customer,
ASIA_Customer;
```

The following example updates column statistics of partitions `USA_Customer`, `EUROPE_Customer` and `ASIA_Customer` for table `Customer`:

```
UPDATE COLUMN STATISTICS FOR Customer FOR PARTITIONS USA_Customer, EUROPE_Customer,
ASIA_Customer;
```

The following example updates all column statistics of partitions `USA_Customer`, `EUROPE_Customer` and `ASIA_Customer` for table `Customer`:

```
UPDATE ALL COLUMN STATISTICS FOR Customer FOR PARTITIONS USA_Customer, EUROPE_Customer,
 ASIA_Customer;
```

The following example shows the commands to get statistics by tenant:

```
UPDATE INDEX STATISTICS FOR table_name FOR TENANTS tenant1, tenant2;
```

The following example shows the commands to update statistics for all CDC system tables:

```
UPDATE STATISTICS FOR CDC SYSTEM TABLES;
```

The following example shows the commands to update specific statistics for all CDC system tables:

```
UPDATE TABLE STATISTICS AND INDEX STATISTICS AND COLUMN STATISTICS FOR CDC SYSTEM TABLES;
```

Authorization

- Must have DBA privilege, SELECT privilege, or ownership of table.

2

OpenEdge SQL functions

This topic provides detailed information on each SQL function. A description for each function provides the following information:

- A definition of the function
- The syntax of the function's proper usage
- A code sample that shows how the function works
- Any associated notes

For details, see the following topics:

- About OpenEdge SQL functions
- ABS
- ACOS
- ADD_MONTHS
- ASCII
- ASIN
- ATAN
- ATAN2
- AVG
- CASE

- CAST
- CDC_get_changed_columns
- CDC_is_column_changed
- CEILING
- CHAR
- CHR
- COALESCE
- CONCAT
- CONVERT (ODBC compatible)
- CONVERT (Progress extension)
- COS
- COUNT
- CURDATE
- CURTIME
- CURRVAL
- DATABASE
- DAYNAME
- DAYOFMONTH
- DAYOFWEEK
- DAYOFYEAR
- DB_NAME
- DECODE
- DEGREES
- EXP
- FLOOR
- GREATEST
- HOUR
- IFNULL
- INITCAP
- INSERT
- INSTR
- ISOWEEKDAY
- ISOWEEK

- ISOYEAR
- LAST_DAY
- LCASE
- LEAST
- LEFT
- LENGTH
- LOCATE
- LOG10
- LOWER
- LPAD
- LTRIM
- MAX
- MIN
- MINUTE
- MOD
- MONTH
- MONTHNAME
- MONTHS_BETWEEN
- NEXT_DAY
- NEXTVAL
- NOW
- NULLIF
- NVL
- PI
- POWER
- PREFIX
- PRO_ARR_DESCAPE function
- PRO_ARR_ESCAPE function
- PRO_ELEMENT function
- QUARTER
- RADIANS
- RAND
- REPEAT

- REPLACE
- RIGHT
- ROUND
- ROWID
- RPAD
- RTRIM
- SECOND
- SIGN
- SIN
- SQRT
- SUBSTR
- SUBSTRING (ODBC compatible)
- SUFFIX
- SUM
- SYSDATE
- SYSTIME
- SYSTIMESTAMP
- TAN
- TO_CHAR
- TO_DATE
- TO_NUMBER
- TO_TIME
- TO_TIMESTAMP
- TRANSLATE
- UCASE
- UPPER
- USER
- WEEK
- YEAR

About OpenEdge SQL functions

A function is an SQL expression that returns a value based on arguments supplied.

OpenEdge® SQL supports five aggregate functions and 90 scalar functions.

Aggregate functions

Aggregate functions calculate a single value for a collection of rows in a result table. If the function is in a statement with a GROUP BY clause, it returns a value for each group in the result table. Aggregate functions are also called set or statistical functions. Aggregate functions cannot be nested. The aggregate functions are:

- AVG
- COUNT
- MAX
- MIN
- SUM

Scalar functions

Scalar functions calculate a value based on another single value. Scalar functions are also called value functions and can be nested.

ABS

Computes the absolute value of *expression*.

Syntax

```
ABS ( expression )
```

Notes

- The argument to the function must be of type TINYINT, SMALLINT, INTEGER, NUMERIC, REAL, or FLOAT.
- The result is of type NUMERIC.
- If the argument *expression* evaluates to NULL, the result is NULL.

Example

This example illustrates the ABS function:

```
SELECT ABS (MONTHS_BETWEEN (SYSDATE, order_date))
    FROM orders
    WHERE  ABS (MONTHS_BETWEEN (SYSDATE, order_date)) > 3 ;
```

Compatibility

ODBC compatible

ACOS

Returns the arccosine of `expression`.

Syntax

```
ACOS ( expression )
```

Notes

- ACOS takes the ratio (*expression*) of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse.
- The result is expressed in radians and is in the range -Pi/2 to Pi/2 radians. To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.
- The expression must be in the range -1 to 1.
- The expression must evaluate to an approximate numeric data type.

Example

In this example, which illustrates two ways to use the ACOS function, the first SELECT statement returns the arcosine in radians, and the second returns the arcosine in degrees:

```
select acos (.5) 'Arccosine in radians' from syscalctable;
ARCCOSINE IN RADIANS
--------------------
1.047197551196598
 1 record selected
  select acos (.5) * (180/ pi()) 'Arccosine in degrees' from syscalctable;
 ARCCOSINE IN DEGREES
--------------------
59.999999999999993
1 record selected
```

Compatibility

ODBC compatible

ADD_MONTHS

Adds to the date value specified by the *date_expression*, the given number of months specified by *integer_expression*, and returns the resultant date value.

Syntax

```
ADD_MONTHS ( date_expression , integer_expression )
```

Notes

- The first argument must be of DATE type.
- The second argument to the function must be of NUMERIC type.
- The result is of type DATE.
- If any of the arguments evaluates to NULL, the result is NULL.

Example

This example illustrates the ADD_MONTHS function:

```
SELECT *
    FROM  customer
    WHERE ADD_MONTHS (start_date, 6) > SYSDATE ;
```

Compatibility

Progress extension

ASCII

Returns the ASCII value of the first character of the given character expression.

Syntax

```
ASCII ( char_expression )
```

Example

The following example shows how to use the ASCII function:

```
SELECT ASCII ( PostalCode )
     FROM Customer;
```

Notes

- The argument to the function must be of type CHARACTER.
- The result is of type INTEGER.
- If the argument *char_expression* evaluates to NULL, the result is NULL.
- The ASCII function is character-set dependent and supports multi-byte characters. The function returns the character encoding integer value of the first character of *char_expression* in the current character set. If *char_expression* is a literal string, the result is determined by the character set of the SQL client. If *char_expression* is a column in the database, the character set of the database determines the result.

Compatibility

ODBC compatible

ASIN

Returns the arcsine of *expression*.

Syntax

```
ASIN ( expression )
```

Notes

- The *expression* must be in the range -1 to 1.
- The *expression* must evaluate to an approximate numeric data type.

Example

In the following example, which shows how to use the ASIN function, the first SELECT statement returns the arcsine in degrees, and the second returns the arcsine in radians:

```
SELECT ASIN (1) * (180/ pi()) 'Arcsine in degrees' FROM SYSPROGRESS.SYSCALCTABLE;
 ARCSINE IN DEGREES
--------------------
90.000000000000000
 1 record selected
  SELECT ASIN (1) 'Arcsine in radians' FROM SYSPROGRESS.SYSCALCTABLE;
 ARCSINE IN RADIANS
--------------------
1.570796326794897
 1 record selected
```

ASIN takes the ratio (*expression*) of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side opposite the angle divided by the length of the hypotenuse.

The result is expressed in radians and is in the range -pi/2 to pi/2 radians. To convert degrees to radians, multiply degrees by pi/180. To convert radians to degrees, multiply radians by 180/pi.

Compatibility

ODBC compatible

ATAN

Returns the arctangent of *expression*.

Syntax

```
ATAN ( expression )
```

Notes

- The *expression* must be in the range -1 to 1.
- The *expression* must evaluate to an approximate numeric data type.

Example

The following example illustrates two ways to use the ATAN function:

```
select atan (1) * (180/ pi()) 'Arctangent in degrees' from syscalctable;
 ARCTANGENT IN DEGREES
----------------------
45.000000000000000
 1 record selected
  select atan (1) 'Arctangent in radians' from syscalctable;
 ARCTANGENT IN RADIANS
----------------------
0.785398163397448
 1 record selected
```

ATAN takes the ratio (*expression*) of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.

The result is expressed in radians and is in the range -Pi/2 to Pi/2 radians. To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.

Compatibility

ODBC compatible

ATAN2

Returns the arctangent of the x and y coordinates specified by expression1 and *expression2*.

Syntax

```
ATAN2 ( expression1 , expression2 )
```

Notes

- ATAN2 takes the ratio of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.
- *expression1* and *expression2* specify the x and y coordinates of the end of the hypotenuse opposite the angle.
- The result is expressed in radians and is in the range -pi/2 to pi/2 radians. To convert degrees to radians, multiply degrees by pi/180. To convert radians to degrees, multiply radians by 180/pi.
- Both *expression1* and *expression2* must evaluate to approximate numeric data types.

Example

The following example illustrates two ways to use the ATAN2 function:

```
select atan2 (1,1) * (180/ pi()) 'Arctangent in degrees' from syscalctable;
 ARCTANGENT IN DEGREES
----------------------
45.000000000000000
 1 record selected
  select atan2 (1,1) 'Arctangent in radians' from syscalctable;
 ARCTANGENT IN RADIANS
----------------------
0.785398163397448
 1 record selected
```

Compatibility

ODBC compatible

AVG

Computes the average of a collection of values. The keyword DISTINCT specifies that the duplicate values are to be eliminated before computing the average.

Syntax

```
AVG ( {[ ALL ] expression }|{ DISTINCT column_ref} )
```

Notes

- NULL values are eliminated before the average value is computed. If all the values are NULL, the result is NULL.
- The argument to the function must be of type SMALLINT, INTEGER, NUMERIC, REAL, or FLOAT.
- The result is of type NUMERIC.

Example

This example illustrates the AVG function:

```
SELECT AVG (salary)
     FROM employee
     WHERE deptno = 20 ;
```

CASE

Specifies a series of search conditions and associated result expressions. The general form is called a searched case expression. SQL returns the value specified by the first result expression whose associated search condition evaluates as true. If none of the search conditions evaluates as true, the CASE expression returns a NULL value, or the value of some other default expression if the CASE expression includes the ELSE clause.

CASE also supports syntax for a shorthand notation, called a simple case expression, for evaluating whether one expression is equal to a series of other expressions.

Syntax

```
searched_case_exprsimple_case_expr
```

Parameters

searched_case_expr

Uses the following syntax:

```
CASE
  WHEN search_condition THEN {result_expr | NULL }[...]
  [ ELSE expr | NULL ]END
```

simple_case_expr

Uses the following syntax:

```
CASE primary_expr  WHEN expr THEN {result_expr | NULL }[...]
  [ ELSE expr | NULL ]
```

CASE

Specifies a searched case expression. It must be followed by one or more WHEN-THEN clauses, each specifying a search condition and corresponding expression.

WHEN *search_condition* THEN { *result_expr* | NULL }

Specifies a search condition and corresponding expression. SQL evaluates *search_condition*. If *search_condition* evaluates as true, CASE returns the value specified by *result_expr*, or NULL, if the clause specifies THEN NULL.

If *search_condition* evaluates as false, SQL evaluates the next WHEN-THEN clause, if any, or the ELSE clause, if it is specified.

CASE *primary_expr*

Specifies a simple case expression. In a simple case expression, one or more WHEN-THEN clauses specify two expressions.

WHEN *expr* THEN {*result_expr* | NULL }

Prompts SQL to evaluate *expr* and compare it with *primary_expr* specified in the CASE clause. If they are equal, CASE returns the value specified by *result_expr* (or NULL, if the clause specifies THEN NULL).

If *expr* is not equal to *primary_expr*, SQL evaluates the next WHEN-THEN clause, if any, or the ELSE clause, if it is specified.

ELSE { *expr* | NULL }

Specifies an optional expression whose value SQL returns if none of the conditions specified in WHEN-THEN clauses are satisfied. If the CASE expression omits the ELSE clause, it is the same as specifying ELSE NULL.

Notes

- This function is not allowed in a GROUP BY clause.
- Arguments to this function cannot be query expressions.

Examples

A simple case expression can always be expressed as a searched case expression. This example illustrates a simple case expression:

```
CASE primary_expr
     WHEN expr1 THEN result_expr1
     WHEN expr2 THEN result_expr2
     ELSE expr3
END
```

The simple case expression in the preceding CASE example is equivalent to the following searched case expression:

```
CASE
     WHEN primary_expr = expr1 THEN result_expr1
     WHEN primary_expr = expr2 THEN result_expr2
     ELSE expr3
END
```

The following example shows a searched case expression that assigns a label denoting suppliers as 'In Mass' if the state column value is 'MA':

```
SELECT name, city,
     CASE
          WHEN state = 'MA' THEN 'In Mass' ELSE 'Not in Mass'
     END
FROM supplier;
 Name             City              searched_case(State,MA,In Mass,)
---------------- ----------------  --------------------------------
GolfWorld Suppl  Boston            In Mass
Pool Swimming S  Valkeala          Not in Mass
Nordic Ski Whol  Hingham           In Mass
Champion Soccer  Harrow            Not in Mass
ABC Sports Supp  Boston            In Mass
Seasonal Sports  Bedford           In Mass
Tennis Supplies  Boston            In Mass
Boating Supplie  Jacksonville      Not in Mass
Aerobic Supplie  Newport Beach     Not in Mass
Sports Unlimite  Irving            Not in Mass
```

The following example shows the equivalent simple case expression:

```
SELECT name, city,
     CASE state
          WHEN 'MA' THEN 'In Mass' ELSE 'Not in Mass'
     END
FROM supplier;
 Name              City               simple_case(State,MA,In Mass,)
---------------- ----------------   ------------------------------
GolfWorld Suppl  Boston             In Mass
Pool Swimming S  Valkeala           Not in Mass
Nordic Ski Whol  Hingham            In Mass
Champion Soccer  Harrow             Not in Mass
ABC Sports Supp  Boston             In Mass
Seasonal Sports  Bedford            In Mass
Tennis Supplies  Boston             In Mass
Boating Supplie  Jacksonville       Not in Mass
Aerobic Supplie  Newport Beach      Not in Mass
Sports Unlimite  Irving             Not in Mass
```

Compatibility

SQL compatible

CAST

Converts an expression to another data type. The first argument is the expression to be converted. The second argument is the target data type.

The length option for the `data_type` argument specifies the length for conversions to `CHAR` and `VARCHAR` data types. If omitted, the default is 1 byte.

If the expression evaluates to `NULL`, the result of the function is null. Specifying `NULL` with the `CAST` function is useful for set operations, such as `UNION`, that require two tables to have the same structure. `CAST NULL` allows you to specify a column of the correct data type, so a table with a similar structure to another, but with fewer columns, can be in a union operation with the other table.

The `CAST` function provides a data-type-conversion mechanism compatible with the SQL standard.

Use the `CONVERT` function, enclosed in the `ODBC` escape clause { fn }, to specify ODBC-compliant syntax for data type conversion. See for more information.

Syntax

```
CAST ( {expression | NULL } AS data_type[ ( length ) ] )
```

Example

The following SQL example uses CAST to convert an integer field from a catalog table to a CHARACTER data type:

```
SELECT CAST(fld AS CHAR(25)), fld FROM sysprogress.syscalctable;
 CONVERT(CHARACTER(25),FLD)     FLD
-------------------------     ---
100                           100
 1 record selected
```

Compatibility

SQL compatible

CDC_get_changed_columns

This scalar function uses change table name and *_Change-FieldMap* column as inputs and returns an array of field names that are changed based on the value of *_Change-FieldMap* column. For each bit whose *_Change-FieldMap* value is set to 1(ON), SQL returns the corresponding field name in the given Change table and forms and array of field names. The entire array is then returned as output.

Note: This function is used on a query which accesses the CDC Change Tracking table, which contains the CDC field map column.

Syntax

```
CDC_get_changed_columns ( pub.CDC_ch1, "_Change-FieldMap" ) ;
```

For example, if pub.CDC_ch1 is the change table and it has fields F1, F2, F3, F4, and F5 and if the _Change-FieldMap column of the Change tracking table contains the values as 11001, then the CDC_get_changed_columns function returns the values F1, F2, and F5.

```
SELECT CDC_get_changed_columns ( "pub"."CDC_mytb14", "_Change-FieldMap" ) FROM
pub."_Cdc-Change-Tracking";
```

Notes

- ‘F1;F2;F5’ is the character representation of the array of field names.
- The JDBC and ODBC drivers do not support array types, and so an array value will be converted to a sequence (or string) of names, each name delimited by a semicolon.

CDC_is_column_changed

This scalar function uses the Change Table name, a field name from the change table and the *_Change-FieldMap* column as inputs and returns the value as either *0* or *1*. If the value returned is *1*, it indicates that the corresponding field value is changed in the FieldMap column. If the value returned is *0*, it indicates that the corresponding field has not changed in the FieldMap column.

For example, if pub.CDC_ch1 is the Change table and it has fields F1, F2, F3, F4, and F5 and the _Change-FieldMap column of the Change Tracking table contains the values as 11001, the following queries to the CDC_get_changed_columns show whether there was a change in the FieldMap column value.

Note: This function is used on a query that accesses the CDC Change Tracking table, which contains the CDC field map column.

Syntax

```
CDC_is_column_changed(pub.CDC_ch1, F1, "_ Change-FieldMap")
```

The above query returns a value of *1* and indicates that the corresponding field value is changed in the FieldMap column.

```
CDC_is_column_changed (pub.CDC_ch1, F3, "_ Change-FieldMap")
```

The above query returns a value of *0* and indicates that the corresponding field value is not changed in the FieldMap column. The following example illustrates this:

```
SELECT "_operation" FROM pub."_CDC-Change-Tracking", WHERE CDC_is_column_changed(
 "pub"."CDC_mytb14", mynum, _Change-FieldMap") = 1;
```

Notes

- SQL returns an error unless the table name given as the first parameter to the function is a Change table.
- The column name given as the second parameter must be a column in that Change table. Otherwise, SQL returns an error

CEILING

Returns the smallest integer greater than or equal to *expression*.

Syntax

```
CEILING ( expression )
```

Notes

The expression must evaluate to a numeric data type.

Example

This example illustrates the CEILING function:

```
SELECT CEILING (32.5) 'Ceiling'
     FROM SYSPROGRESS.SYSCALCTABLE;
```

Compatibility

ODBC compatible

CHAR

Returns a character string with the first character having an ASCII value equal to the argument expression. CHAR is identical to CHR but provides ODBC-compatible syntax.

Syntax

```
CHAR ( integer_expression )
```

Notes

- The argument to the function must be of type INTEGER, TINYINT, or SMALLINT.
- The result is of type CHARACTER.
- If the argument *integer_expression* evaluates to NULL, the result is NULL.
- The CHAR and CHR functions are character-set dependent and support single-byte and multi-byte characters. If *integer_expression* is a valid character encoding integer value in the current SQL server character set, the function returns the correct character. If it is not a valid character, the function returns a NULL value.

Example

This example illustrates the CHAR function:

```
SELECT *
    FROM  customer
    WHERE SUBSTR (zip, 1, 1) = CHAR (53) ;
```

Compatibility

ODBC compatible

CHR

Returns a character string with the first character having an ASCII value equal to the argument expression.

Syntax

```
CHR ( integer_expression )
```

Notes

- The argument to the function must be of type INTEGER, TINYINT, or SMALLINT.
- The result is of type CHARACTER.
- If the argument *integer_expression* evaluates to NULL, the result is NULL.
- The CHR and CHAR functions are character-set dependent, and support multi-byte characters. If *integer_expression* is a valid character encoding integer value in the current SQL server character set, the function returns the correct character. If it is not a valid character the function returns a NULL value.

Example

This example illustrates the CHR function and the SUBSTR (substring) function:

```
SELECT *
    FROM  customer
    WHERE SUBSTR (zip, 1, 1) = CHR (53) ;
```

Compatibility

Progress extension

COALESCE

Specifies a series of expressions and returns the first expression whose value is not NULL. If all the expressions evaluate as null, COALESCE returns a NULL value.

Syntax

```
COALESCE(expression1, expression2[...])
```

The COALESCE syntax is shorthand notation for a common case that can also be represented in a CASE expression. The following two formulations are equivalent:

```
COALESCE ( expression1 , expression2 , expression3 )
```

```
CASE
    WHEN expression1 IS NOT NULL THEN expression1
    WHEN expression2 IS NOT NULL THEN expression2
    ELSE expression3
END
```

Notes

- This function is not allowed in a GROUP BY clause.
- Arguments to this function cannot be query expressions.

Example

This example illustrates the COALESCE function:

```
SELECT COALESCE (end_date, start_date) from job_hist;
```

Compatibility

SQL compatible

CONCAT

Returns a concatenated character string formed by concatenating two arguments.

Syntax

```
CONCAT ( char_expression , char_expression )
```

Notes

- Both of the arguments must be of type CHARACTER or VARCHAR.
- The result is of type VARCHAR.
- If any of the argument expressions evaluate to NULL, the result is NULL.
- The two *char_expression* expressions and the result of the CONCAT function can contain multi-byte characters.

Example

This example illustrates the CONCAT function:

```
SELECT last_name, empno, salary
    FROM customer
    WHERE project = CONCAT('US',proj_nam);
```

Compatibility

ODBC compatible

CONVERT (ODBC compatible)

Converts an expression to another data type. The first argument is the expression to be converted. The second argument is the target data type.

If the expression evaluates to NULL, the result of the function is NULL.

The ODBC CONVERT function provides ODBC-compliant syntax for data type conversions. You must enclose the function with the ODBC escape clause { fn } to use ODBC-compliant syntax.

Syntax

```
{ fn CONVERT ( expression , data_type ) }
```

Notes

Braces are part of the actual syntax. The following data types are used:

```
SQL_BINARY | SQL_CHAR | SQL_DATE | SQL_DECIMAL
   | SQL_DOUBLE | SQL_FLOAT | SQL_INTEGER | SQL_REAL
   | SQL_SMALLINT | SQL_TIME | SQL_TIMESTAMP | SQL_TINYINT
   | SQL_VARBINARY | SQL_VARCHAR
```

Compatibility

ODBC compatible

CONVERT (Progress extension)

Converts an expression to another data type. The first argument is the target data type. The second argument is the expression to be converted to that type.

The length option for the *data_type* argument specifies the length for conversions to CHAR and VARCHAR data types. If omitted, the default is 30 bytes.

If the expression evaluates to NULL, the result of the function is NULL.

The CONVERT function syntax is similar to, but not compatible with, the ODBC CONVERT function. Enclose the function in the ODBC escape clause { fn } to specify ODBC-compliant syntax. See the ODBC compatible CONVERT function for more information.

Syntax

```
CONVERT ( 'data_type [ ( length ) ]', expression )
```

Notes

When *data_type* is CHARACTER(*length*) or VARCHAR(*length*), the *length* specification represents the number of characters. The converted result can contain multi-byte characters.

Example

The following SQL example uses the CONVERT function to convert an INTEGER field from a system table to a character string:

```
SELECT CONVERT('CHAR', fld), fld FROM sysprogress.syscalctable;
 CONVERT(CHAR,FLD)               FLD
-----------------               ---
100                             100
 1 record selected
  SELECT CONVERT('CHAR(35)', fld), fld FROM sysprogress.syscalctable;
 CONVERT(CHAR(35),FLD)              FLD
---------------------              ---
100                                100
 1 record selected
```

Compatibility

Progress extension

COS

Returns the cosine of *expression*.

Syntax

```
COS ( expression )
```

Notes

- COS takes an angle expression and returns the ratio of two sides of a right triangle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse.
- The expression specifies an angle in radians.
- The expression must evaluate to an approximate numeric data type.
- To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.

Example

This example illustrates the COS function:

```
select cos(45 * pi()/180) 'Cosine of 45 degrees'
     from sysprogress.syscalctable;
 COSINE OF 45 DEG
----------------
0.707106781186548
 1 record selected
```

Compatibility

ODBC compatible

COUNT

Computes either the number of rows in a group of rows or the number of non-NULL values in a group of values.

Syntax

```
COUNT ( {[ ALL ]expression}|{ DISTINCT column_ref}| * )
```

Notes

- The keyword DISTINCT specifies that the duplicate values are to be eliminated before computing the count.
- If the argument to COUNT function is '*', then the function computes the count of the number of rows in a group.
- If the argument to COUNT function is not '*', then NULL values are eliminated before the number of rows is computed.
- The argument *column_ref* or *expression* can be of any type.
- The result of the function is of BIGINT data type. The result is never NULL.

Example

This example illustrates the COUNT function:

```
SELECT COUNT (*)
     FROM orders
     WHERE order_date = SYSDATE ;
```

CURDATE

Returns the current date as a DATE value. This function takes no arguments.

Syntax

```
CURDATE ( )
```

Notes

SQL statements can refer to CURDATE anywhere they can refer to a DATE expression.

Example

The following example shows how to use the CURDATE function:

```
INSERT INTO objects (object_owner, object_id, create_date)
     VALUES (USER, 1001, CURDATE()) ;
```

Compatibility

ODBC compatible

CURTIME

Returns the current time as a TIME value. This function takes no arguments.

Syntax

```
CURTIME ( )
```

Notes

SQL statements can refer to CURTIME anywhere they can refer to a TIME expression.

Example

This example illustrates how to use the CURTIME function to INSERT the current time into the create_time column of the objects table:

```
INSERT INTO objects (object_owner, object_id, create_time)
     VALUES (USER, 1001, CURTIME()) ;
```

Compatibility

ODBC compatible

CURRVAL

CURRVAL returns the current value of a sequence, and uses the following syntax to reference the current value of a sequence.

Syntax

```
schema.sequence. CURRVAL
```

Parameters

schema

Specifies the schema that contains the sequence. To refer to the current value of a sequence in the schema of another user, you must have SELECT object privilege on the sequence.

sequence

Specifies the name of the sequence whose current value you want.

Use CURRVAL in:

- The SELECT list of a SELECT statement not contained in a subquery or view
- The SELECT list of a subquery in an INSERT statement
- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

CURRVAL cannot be used in:

- A query of a view
- A SELECT statement with a GROUP BY clause that references a sequence
- A SELECT statement with an ORDER BY clause that references a sequence
- A SELECT statement that is combined with another SELECT statement with the UNION, INTERSECT, or MINUS set operator
- The WHERE clause of a SELECT or UPDATE statement
- The DEFAULT value of a column in a CREATE TABLE or ALTER TABLE statement
- The condition of a CHECK constraint

Example

In the following example, the OpenEdge SQL sequence generator returns the current value of the customer sequence:

```
SELECT customer_sequence.CURRVAL FROM pub.customer;
```

DATABASE

Returns the name of the database corresponding to the current connection name. This function takes no arguments, and the trailing parentheses are optional.

Syntax

```
DATABASE [ ( ) ]
```

Example

The following example shows how to use the DATABASE function:

```
select database() from t2;
 DATABASE
--------
steel
 1 record selected
```

Compatibility

ODBC compatible

DAYNAME

Returns a character string containing the name of the day (for example, Sunday through Saturday) for the day portion of *date_expression*. The argument *date_expression* can be the name of a column, the result of another scalar function, or a date or timestamp literal.

Syntax

```
DAYNAME ( date_expression )
```

Example

This example illustrates the DAYNAME function:

```
SELECT *
     FROM orders
     WHERE order_no = 342 and DAYNAME(order_date)= 'SATURDAY';
  ORDER_NO  ORDER_DATE  REFERENCE         CUST_NO
-------- -----------  ----------        --------
342       08/10       tdfg/101          10001
 1 record selected
```

Compatibility

ODBC compatible

DAYOFMONTH

Returns the day of the month in the argument as a short integer value in the range of 1-31. The argument *date_expression* can be the name of a column, the result of another scalar function, or a date or timestamp literal.

Syntax

```
DAYOFMONTH ( date_expression )
```

Notes

- The *date_expression* argument must be of type DATE.
- If *date_expression*is supplied as a date literal, it can be any of the valid *date_literal*formats where the day specification (DD) precedes the month specification (MM).
- The result is of type SHORT.
- If the argument expression evaluates to NULL, the result is NULL.

Example

This example illustrates the DAYOFMONTH function:

```
SELECT *
    FROM orders
    WHERE DAYOFMONTH (order_date) = 14 ;
```

Compatibility

ODBC compatible

DAYOFWEEK

Returns the day of the week in the argument as a short integer value in the range of 1-7.

The argument date_expression can be the name of a column, the result of another scalar function, or a date or timestamp literal.

Syntax

```
DAYOFWEEK ( date_expression )
```

Notes

- The argument to the function must be of type DATE.
- If *date_expression* is supplied as a date literal, it can be any of the valid *date_literal*formats where the day specification (DD) precedes the month specification (MM).

- The result is of type SHORT.
- If the argument expression evaluates to NULL, the result is NULL.

Example

The following example shows how to use the DAYOFWEEK function:

```
SELECT *
    FROM orders
    WHERE DAYOFWEEK (order_date) = 2 ;
```

Compatibility

ODBC compatible

DAYOFYEAR

Returns the day of the year in the argument as a short integer value in the range of 1-366. The argument *date_expression* can be the name of a column, the result of another scalar function, or a date or timestamp literal.

Syntax

```
DAYOFYEAR ( date_expression )
```

Notes

- The argument to the function must be of type DATE.
- The result is of type SHORT.
- If the argument expression evaluates to NULL, the result is NULL.

Example

This example illustrates the DAYOFYEAR function:

```
SELECT *
    FROM orders
    WHERE DAYOFYEAR (order_date) = 300 ;
```

Compatibility

ODBC compatible

DB_NAME

Returns the name of the database corresponding to the current connection name.

Syntax

```
DB_NAME ( )
```

Example

This example illustrates the DB_NAME function:

```
SELECT DB_NAME() FROM T2;
 DB_NAME
-------
demo
 1 record selected
```

Compatibility

Progress extension

DECODE

Compares the value of the first argument *expression* with each *search_expression* and, if a match is found, returns the corresponding *match_expression*. If no match is found, then the function returns the *default_expression*. If a *default_expression* is not specified and no match is found, then the function returns a NULL value.

Syntax

```
DECODE ( expression, search_expression, match_expression
  [ , search_expression, match_expression... ]
  [ , default_expression ] )
```

Example

This example illustrates one way to use the DECODE function:

```
SELECT ename, DECODE (deptno,
             10, 'ACCOUNTS',
             20, 'RESEARCH',
             30, 'SALES',
             40, 'SUPPORT',
             'NOT ASSIGNED'
          )
   FROM employee ;
```

Notes

- Use a simple case expression when SQL-compatible syntax is a requirement.

- The first argument *expression* can be of any type. The types of all *search_expressions* must be compatible with the type of the first argument.
- The *match_expressions* can be of any type. The types of all *match_expressions* must be compatible with the type of the first *match_expression*.
- The type of the *default_expression* must be compatible with the type of the first *match_expression*.
- The type of the result is the same as that of the first *match_expression*.
- If the first argument *expression* is NULL, then the value of the *default_expression* is returned, if it is specified. Otherwise NULL is returned.

Compatibility

Progress extension

DEGREES

Returns the number of degrees in an angle specified in radians by *expression*.

Syntax

```
DEGREES ( expression )
```

Notes

- The *expression* specifies an angle in radians.
- The *expression* must evaluate to a numeric data type.

Example

This example illustrates the DEGREES function:

```
SELECT DEGREES(3.14159265359) 'Degrees in pi Radians'
     FROM SYSPROGRESS.SYSCALCTABLE;
```

Compatibility

ODBC compatible

EXP

Returns the exponential value of expression (*e* raised to the power of expression).

Syntax

```
EXP ( expression )
```

Notes

expression must evaluate to an approximate numeric data type.

Example

This example illustrates the EXP function:

```
SELECT EXP( 4 ) 'e to the 4th power' from sysprogress.syscalctable;
```

Compatibility

ODBC compatible

FLOOR

Returns the largest integer less than or equal to `expression`.

Syntax

```
FLOOR ( expression )
```

Notes

expression must evaluate to a numeric data type.

Example

This example illustrates the FLOOR function:

```
SELECT FLOOR (32.5) 'Floor' from sysprogress.syscalctable ;
```

Compatibility

ODBC compatible

GREATEST

Returns the greatest value among the values of the given expressions.

Syntax

```
GREATEST ( expression , expression... )
```

Notes

- The first argument to the function can be of any type. However, the types of the subsequent arguments must be compatible with that of the first argument.
- The type of the result is the same as that of the first argument.
- If any of the argument expressions evaluate to NULL, the result is NULL.
- When the data type of an *expression* is either CHARACTER(*length*) or VARCHAR(*length*), the expression can contain multi-byte characters. The sort weight for each character is determined by the collation table in the database.

Example

This example illustrates the GREATEST function:

```
SELECT cust_no, last_name,
    GREATEST (ADD_MONTHS (start_date, 10), SYSDATE)
    FROM customer ;
```

Compatibility

Progress extension

HOUR

Returns the hour in the argument as a short integer value in the range of 0-23.

Syntax

```
HOUR ( time_expression )
```

Notes

- The argument to the function must be of type TIME.
- The argument must be specified in the format hh:mi:ss.
- The result is of type SHORT.
- If the argument expression evaluates to NULL, the result is NULL.

Example

This example illustrates the HOUR function:

```
SELECT *
    FROM arrivals
    WHERE HOUR (in_time) < 12 ;
```

Compatibility

ODBC compatible

IFNULL

Returns *value* if *expr* is NULL. If *expr* is not NULL, IFNULL returns *expr*.

Syntax

```
IFNULL( expr, value)
```

Notes

The data type of *value* must be compatible with the data type of *expr*.

Example

In this example, which illustrates the IFNULL function, the SELECT statement returns three rows with a NULL value in column C1, and two non-NULL values:

```
SELECT C1, IFNULL(C1, 9999) FROM TEMP ORDER BY C1;
 C1    IFNULL(C1,9999)
--    ----------------
      9999
      9999
      9999
1     1
3     3
```

Compatibility

ODBC compatible

INITCAP

Returns the result of the argument character expression after converting the first character to uppercase and the subsequent characters to lowercase.

Syntax

```
INITCAP ( char_expression )
```

Notes

- The *char_expression* must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to NULL, the result is NULL.

- A *char_expression* and the result can contain multi-byte characters. The uppercase conversion for the first character and the lowercase conversion for the rest of the characters is based on the case table in the *convmap* file. The default case table is BASIC.

Example

The following example shows how to use the INITCAP function:

```
SELECT INITCAP (last_name)
     FROM customer ;
```

Compatibility

Progress extension

INSERT

Returns a character string where length number of characters have been deleted from string_exp1 beginning at start_pos, and *string_exp2* has been inserted into *string_exp1*, beginning at start_pos.

Syntax

```
INSERT( string_exp1 , start_pos , length , string_exp2 )
```

Notes

- The *string_exp* can be type fixed-length or variable-length CHARACTER.
- The *start_pos* and *length* can be of data type INTEGER, SMALLINT, or TINYINT.
- The result string is of the type *string_exp1*.
- If any of the argument expressions evaluate to NULL, the result is NULL.
- If *start_pos* is negative or zero, the result string evaluates to NULL.
- If *length* is negative, the result evaluates to NULL.
- *string_exp1* and *string_exp2* and the result can contain multi-byte characters. This is determined by the character set of the SQL server. The length argument specifies a number of characters.

Example

This example illustrates the INSERT function:

```
SELECT INSERT(last_name,2,4,'xx')
     FROM customer
     WHERE last_name = 'Goldman';
 INSERT LAST_NAME,2,4,XX)
------------------------
Gxxan
 1 record selected
```

The two letters 'o' and 'l' are deleted from the name 'Goldman' in the last_name column, and the letters 'xx' are inserted into the last_name column, beginning at the fourth character, overlaying the letters 'd' and 'm'.

Compatibility

ODBC compatible

INSTR

Searches character string *char_expression1* for the character string *char_expression2*. The search begins at *start_pos* of *char_expression1*. If *occurrence* is specified, then INSTR searches for the *n*th occurrence, where *n* is the value of the fourth argument.

The position (with respect to the start of char_expression1) is returned if a search is successful. Zero is returned if no match can be found.

Syntax

```
INSTR ( char_expression1 , char_expression2[ , start_pos[ , occurrence]] )
```

Notes

- The first and second arguments must be CHARACTER data type.
- The third and fourth arguments, if specified, must be SMALLINT or TINYINT data type.
- The value for start position in a character string is the ordinal number of the character in the string. The very first character in a string is at position 1, the second character is at position 2, the nth character is at position n.
- If you do not specify *start_pos*, a default value of 1 is assumed.
- If you do not specify *occurrence*, a default value of 1 is assumed.
- The result is INTEGER data type.
- If any of the argument expressions evaluate to NULL, the result is NULL.
- *A char_expression* and the result can contain multi-byte characters.

Example

This example illustrates the INSTR function:

```
SELECT cust_no, last_name
    FROM customer
    WHERE INSTR (LOWER (addr), 'heritage') > 0 ;
```

Compatibility

Progress extension

ISOWEEKDAY

Returns the Day of the ISO Week, based on ISO 8601 standards, as a short integer value in the range of 1-7.

Syntax

```
ISOWEEKDAY ( time_expression )
```

Notes

- The argument to the function must be of type DATE.
- If *date_expression* is supplied as a date literal, it can be any of the valid *date_literal* formats where the day specification (DD) precedes the month specification (MM).
- The result is of type SHORT.
- If the argument expression evaluates to NULL, the result is NULL.

Example

The query returns all columns from rows in the orders table where the order_date is in the fifth day of the ISO week. The following examples illustrate the ISOWEEKDAY function:

```
SELECT *
    FROM orders
    WHERE ISOWEEKDAY (order_date) = 5 ;
```

Compatibility

ODBC compatible

ISOWEEK

Returns the ISO WEEK of the year, based on ISO 8601 standards, as a short integer value in the range of 1-53.

Syntax

```
ISOWEEK ( time_expression )
```

Notes

- The argument to the function must be of type DATE.
- If *date_expression* is supplied as a date literal, it can be any of the valid *date_literal* formats where the day specification (DD) precedes the month specification (MM).
- The result is of type SHORT.

- If the argument expression evaluates to NULL, the result is NULL.

Example

The query returns all columns from rows in the orders table where the order_date is in the fifth ISO week of the year. This example illustrates the ISOWEEK function:

```
SELECT *
    FROM orders
    WHERE ISOWEEK (order_date) = 5 ;
```

Compatibility

ODBC compatible

ISOYEAR

Returns the ISO Year, based on ISO 8601 standards, as a short integer value in the range of 0-9999.

Syntax

```
ISOYEAR ( time_expression )
```

Notes

- The argument to the function must be of type DATE.
- If *date_expression* is supplied as a date literal, it can be any of the valid *date_literal* formats where the day specification (DD) precedes the month specification (MM).
- The result is of type SHORT.
- If the argument expression evaluates to NULL, the result is NULL.

Example

The query returns all columns from rows in the orders table where the ISO year of order_date is 2020. This example illustrates the ISOYEAR function:

```
SELECT *
    FROM orders
    WHERE ISOYEAR (order_date) = 2020 ;
```

Compatibility

ODBC compatible

For more information, see YEAR.

LAST_DAY

Returns the date corresponding to the last day of the month containing the argument date.

Syntax

```
LAST_DAY ( date_expression )
```

Notes

- The argument to the function must be of type DATE.
- The result is of type DATE.
- If the argument expression evaluates to NULL, the result is NULL.

Example

This example illustrates the LAST_DAY function:

```
SELECT *
    FROM orders
    WHERE LAST_DAY (order_date) + 1 = '08/01/2003' ;
```

Compatibility

Progress extension

LCASE

Returns the result of the argument character expression after converting all the characters to lowercase. LCASE is the same as LOWER but provides ODBC-compatible syntax.

Syntax

```
LCASE ( char_expression )
```

Notes

- The argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to NULL, the result is NULL.
- A *char_expression* and the result can contain multi-byte characters. The lowercase conversion is determined by the case table in the convmap file. The default case table is BASIC.

Example

This example illustrates the LCASE function:

```
SELECT *
    FROM customer
    WHERE LCASE (last_name) = 'smith' ;
```

Compatibility

ODBC compatible

LEAST

Returns the lowest value among the values of the given expressions.

Syntax

```
LEAST ( expression , expression, ... )
```

Notes

- The first argument to the function can be of any type. However, the types of the subsequent arguments must be compatible with that of the first argument.
- The type of the result is the same as that of the first argument.
- If any of the argument expressions evaluate to NULL, the result is NULL.
- When the data type of an *expression*is either CHARACTER(*length*) or VARCHAR(*length*), the *expression*can contain multi-byte characters. The sort weight for each character is determined by the collation table in the database.

Example

This example illustrates the LEAST function:

```
SELECT cust_no, last_name,
     LEAST (ADD_MONTHS (start_date, 10), SYSDATE)
     FROM customer ;
```

Compatibility

Progress extension

LEFT

Returns the leftmost count of characters of *string_exp*.

Syntax

```
LEFT ( string_exp , count )
```

Notes

- *string_exp* can be fixed-length or variable-length CHARACTER data types.
- *count* can be INTEGER, SMALLINT, or TINYINT data types.
- If any of the arguments of the expression evaluate to NULL, the result is NULL.
- If the *count* is negative, the result evaluates to NULL.
- The *string_exp* and the result can contain multi-byte characters. The function returns the number of characters.

Example

The following example shows how to use the LEFT function:

```
SELECT LEFT(last_name,4) FROM  customer WHERE last_name = 'Goldman';
 LEFT(LAST_NAME),4)
------------------
Gold
 1 record selected
```

Compatibility

ODBC compatible

LENGTH

Returns the string length of the value of the given character expression.

Syntax

```
LENGTH ( char_expression )
```

Notes

- The argument to the function must be of type CHARACTER or VARCHAR.
- The result is of type INTEGER.
- If the argument expression evaluates to NULL, the result is NULL.
- *char_expression* can contain multi-byte characters. The function returns a number of characters.

Example

This example illustrates the LENGTH function:

```
SELECT last_name 'LONG LAST_NAME'
     FROM customer
     WHERE LENGTH (last_name) > 5 ;
```

Compatibility

ODBC compatible

LOCATE

Returns the location of the first occurrence of char_expr1 in char_expr2. If the function includes the optional integer argument start_pos, LOCATE begins searching char_expr2 at that position. If the function omits the start_pos argument, LOCATE begins its search at the beginning of char_expr2.

LOCATE denotes the first character position of a character expression as 1. If the search fails, LOCATE returns 0. If either character expression is NULL, LOCATE returns a NULL value.

Syntax

```
LOCATE( char_expr1 , char_expr2 , [start_pos] )
```

Notes

char_expr1 and *char_expr2* can contain multi-byte characters. The *start_pos* argument specifies the position of a starting character, not a byte position. The search is case sensitive. Character comparisons use the collation table in the database.

Example

In the following example, which uses two string literals as character expressions, LOCATE returns a value of 6:

```
SELECT LOCATE('this', 'test this test', 1) FROM TEST;
 LOCATE(THIS,
------------
61 record selected
```

Compatibility

ODBC compatible

LOG10

Returns the base 10 logarithm of expression.

Syntax

```
LOG10 ( expression )
```

Example

This example illustrates the LOG10 function:

```
SELECT LOG10 (100) 'Log base 10 of 100' FROM SYSPROGRESS.SYSCALCTABLE;
```

Notes

The *expression* must evaluate to an approximate numeric data type.

Compatibility

ODBC compatible

LOWER

Returns the result of the argument char_expression after converting all the characters to lowercase.

Syntax

```
LOWER ( char_expression )
```

Example

This example illustrates the LOWER function:

```
SELECT *
    FROM customer
    WHERE LOWER (last_name) = 'smith' ;
```

Notes

- The argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to NULL, the result is NULL.

Compatibility

SQL compatible

LPAD

Pads the character string corresponding to the first argument on the left with the character string corresponding to the third argument. After the padding, the length of the result is *length*.

Syntax

```
LPAD ( char_expression , length [ , pad_expression ] )
```

Notes

- The first argument to the function must be of type CHARACTER. The second argument to the function must be of type INTEGER. The third argument, if specified, must be of type CHARACTER. If the third argument is not specified, the default value is a string of length 1 containing one blank.
- If L1 is the length of the first argument and *L2* is the value of the second argument:
  - If L1 is less than *L2*, the number of characters padded is equal to *L2* minus *L1*.
  - If L1 is equal to *L2*, no characters are padded and the result string is the same as the first argument.
  - If L1 is greater than *L2*, the result string is equal to the first argument truncated to the first *L2* characters.
- The result is of type CHARACTER.
- If the argument expression evaluates to NULL, the result is NULL.
- The *char_expression* and *pad_expression* can contain multi-byte characters. The *length* specifies a number of characters.

Example

This example illustrates two ways to use the LPAD function:

```
SELECT LPAD (last_name, 30) FROM customer ;
 SELECT LPAD (last_name, 30, '.') FROM customer ;
```

Compatibility

Progress extension

LTRIM

Removes all the leading characters in *char_expression* that are present in *char_set* and returns the resulting string. The first character in the result is guaranteed not to be in *char_set*. If you do not specify the *char_set* argument, leading blanks are removed.

Syntax

```
LTRIM ( char_expression[ , char_set] )
```

Notes

- The first argument to the function must be of type CHARACTER.
- The second argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to NULL, the result is NULL.
- The *char_expression*, the character set specified by *char_set*, and the result can contain multi-byte characters.

Example

This example illustrates the LTRIM function:

```
SELECT last_name, LTRIM (addr, ' ')
     FROM customer ;
```

Compatibility

ODBC compatible

MAX

Returns the maximum value in a group of values.

Syntax

```
COUNT ( {[ ALL ]expression}|{ DISTINCT column_ref}| * )
```

Notes

- Specifying DISTINCT has no effect on the result.
- The argument *column_ref* or *expression* can be of any type.
- The result of the function is of the same data type as that of the argument.
- The result is NULL if the result set is empty or contains only NULL values.

Example

This example illustrates the MAX function:

```
SELECT order_date, product, MAX (qty)
     FROM orders
     GROUP BY order_date, product ;
```

MIN

Returns the minimum value in a group of values.

Syntax

```
MIN ( {[ ALL ]expression}|{ DISTINCT column_ref} )
```

Notes

- Specifying DISTINCT has no effect on the result.
- The argument *column_ref* or *expression* can be of any type.
- The result of the function is of the same data type as that of the argument.
- The result is NULL if the result set is empty or contains only NULL values.

Example

This example illustrates the MIN function:

```
SELECT MIN (salary)
     FROM employee
     WHERE deptno = 20 ;
```

MINUTE

Returns the minute value in the argument as a short integer in the range of 0-59.

Syntax

```
MINUTE ( time_expression )
```

Notes

- The argument to the function must be of type TIME.
- The argument must be specified in the format *HH:MI:SS*.

- The result is of type SHORT.
- If the argument expression evaluates to NULL, the result is NULL.

Example

This example illustrates the MINUTE function:

```
SELECT *
     FROM arrivals
     WHERE MINUTE (in_time) > 10 ;
```

Compatibility

ODBC compatible

MOD

Returns the remainder of expression1 divided by expression2.

Syntax

```
MOD ( expression1 , expression2 )
```

Notes

- Both *expression1* and *expression2* must evaluate to exact numeric data types.
- If *expression2* evaluates to zero, MOD returns zero.

Example

This example illustrates the MOD function:

```
SELECT MOD (11, 4) 'Modulus' FROM MYMATH;
```

Compatibility

ODBC compatible

MONTH

Returns the month in the year specified by the argument as a short integer value in the range of 1-12.

Syntax

```
MONTH ( date_expression )
```

Notes

- The argument to the function must be of type DATE.
- If *date_expression* is supplied as a time literal, it can be any of the valid *date_literal* formats where the day specification (DD) precedes the month specification (MM).
- The result is of type SHORT.
- If the argument expression evaluates to NULL, the result is NULL.

Example

This example illustrates the MONTH function:

```
SELECT * FROM orders WHERE MONTH (order_date) = 6 ;
```

Compatibility

ODBC compatible

MONTHNAME

Returns a character string containing the name of the month (for example, January through December) for the month portion of *date_expression*. The argument *date_expression* can be the name of a column, the result of another scalar function, or a date or timestamp literal.

Syntax

```
MONTHNAME ( date_expression )
```

Example

In this example, which illustrates the MONTHNAME function, the query returns all rows where the name of the month in the order_date column is equal to 'June':

```
SELECT *
FROM orders
WHERE order_no =346 and MONTHNAME(order_date)='JUNE';
 ORDER_NO ORDER_DATE  REFERENCE       CUST_NO
------- ---------   --------         -------
346     06/01/2003 87/rd          10002
 1 record selected
```

Compatibility

ODBC compatible

MONTHS_BETWEEN

Computes the number of months between two date values corresponding to the first and second arguments.

Syntax

```
MONTHS_BETWEEN ( date_expression, date_expression )
```

Notes

- The first and second arguments to the function must be of type DATE.
- The result is of type INTEGER.
- The result is negative if the date corresponding to the second argument is greater than that corresponding to the first argument.
- If any of the argument expressions evaluates to NULL, the result is NULL.

Example

This example illustrates the MONTHS_BETWEEN function:

```
SELECT MONTHS_BETWEEN (SYSDATE, order_date)
    FROM orders
    WHERE order_no = 1002 ;
```

Compatibility

Progress extension

NEXT_DAY

Returns the minimum date that is greater than the date corresponding to the first argument where the day of the week is the same as that specified by the second argument.

Syntax

```
NEXT_DAY ( date_expression, day_of_week )
```

Notes

- The first argument to the function must be of type DATE.
- The second argument to the function must be of type CHARACTER. The result of the second argument must be a valid day of the week (‘SUNDAY', ‘MONDAY' etc.).
- The result is of type DATE.
- If any of the argument expressions evaluate to NULL, the result is NULL.

Example

This example illustrates the NEXT_DAY function:

```
SELECT NEXT_DAY (order_date, 'MONDAY') FROM orders ;
```

Compatibility

Progress extension

NEXTVAL

NEXTVAL returns a sequence's next value. References to NEXTVAL increment the sequence value by the defined increment and return the new value.

Use the following syntax to reference the next value of a sequence:

Syntax

```
schema.sequence.NEXTVAL
```

Parameters

schema

Specifies the schema that contains the sequence. To refer to the next value of a sequence in the schema of another user, you must have SELECT object privilege on the sequence.

sequence

Specifies the name of the sequence whose next value you want.A statement referencing NEXTVAL for a noncycling sequence returns an error after reaching the maximum value.

Use NEXTVAL in the:

- SELECT list of a SELECT statement not contained in a subquery or view
- SELECT list of a subquery in an INSERT statement
- VALUES clause of an INSERT statement
- SET clause of an UPDATE statement

NEXTVAL cannot be used in:

- A query of a view
- A SELECT statement with a GROUP BY clause that references a sequence
- A SELECT statement with an ORDER BY clause that references a sequence
- A SELECT statement that is combined with another SELECT statement with the UNION, INTERSECT, or MINUS set operator and this restriction applies even if UNION, INTERSECT, or MINUS occur in a derived table clause
- The WHERE clause of a SELECT or UPDATE statement

- The DEFAULT value of a column in a CREATE TABLE or ALTER TABLE statement
- The condition of a CHECK constraint

Example

In the following example, the sequence generator increments the customer sequence and uses its value for a new customer inserted into the table pub.customer:

```
INSERT INTO pub.customer VALUES (customer_sequence.NEXTVAL,'USA','BackCountry
Equipment','Sugar Hill Road','12A','Franconia','NH','03242','Dan
Egan','603-762-2121','Kirsten Ulmner', 10000.00, 500.00,'net 10', 0,'contact monthly');
```

NOW

Returns the current date and time as a TIMESTAMP value. This function takes no arguments.

Syntax

```
NOW ( )
```

Compatibility

ODBC compatible

NULLIF

Returns a NULL value for *expression1* if it is equal to *expression2*. It is useful for converting values to NULL from applications that use some other representation for missing or unknown data. The NULLIF scalar function is a type of conditional expression.

Syntax

```
NULLIF ( expression1, expression2 )
```

Notes

- This function is not allowed in a GROUP BY clause.
- Arguments to this function cannot be query expressions.
- The NULLIF expression is shorthand notation for a common case that can also be represented in a CASE expression, as shown:

```
CASE
    WHEN expression1 = expression2 THEN NULL
    ELSE expression1
END
```

Example

This example uses the NULLIF scalar function to insert a NULL value into an address column if the host-language variable contains a single space character:

```
INSERT INTO employee (add1) VALUES (NULLIF (:address1, ' '));
```

Compatibility

SQL compatible

NVL

Returns the value of the first expression if the first expression value is not NULL. If the first expression value is NULL, the value of the second expression is returned.

Syntax

```
NVL ( expression , expression )
```

Notes

- The NVL function is not ODBC compatible. Use the IFNULL function when ODBC-compatible syntax is required.
- The first argument to the function can be of any type.
- The type of the second argument must be compatible with that of the first argument.
- The type of the result is the same as the first argument.

Example

This example illustrates the NVL function:

```
SELECT salary + NVL (comm, 0) 'TOTAL SALARY' FROM employee ;
```

Compatibility

Progress extension

PI

Returns the constant value of PI as a floating-point value.

Syntax

```
PI ( )
```

Example

This example illustrates the PI function:

```
SELECT PI () FROM SYSPROGRESS.SYSCALCTABLE;
```

Compatibility

ODBC compatible

POWER

Returns *expression1* raised to the power of *expression2*.

Syntax

```
POWER ( expression1 , expression2 )
```

Notes

- *expression1* must evaluate to a numeric data type.
- *expression2* must evaluate to an exact numeric data type.

Example

This example illustrates the POWER function, raising ‘3' to the second power:

```
SELECT POWER ( 3 , 2) '3 raised to the 2nd power'
     FROM SYSPROGRESS.SYSCALCTABLE;
```

PREFIX

Returns the substring of a character string, starting from the position specified by *start_pos* and ending before the specified character.

Syntax

```
PREFIX ( char_expression , start_pos , char_expression )
```

Parameters

char_expression

Evaluates to a character string, typically a character-string literal or column name. If the expression evaluates to NULL, PREFIX returns NULL.

start_pos

Evaluates to an integer value. PREFIX searches the string specified in the first argument starting at that position. A value of 1 indicates the first character of the string.

char_expression

Evaluates to a single character. PREFIX returns the substring that ends before that character. If PREFIX does not find the character, it returns the substring beginning at *start_pos*, to the end of the string. If the expression evaluates to more than one character, PREFIX ignores all but the first character.

Notes

Each *char_expression* and the result can contain multi-byte characters. The start_pos argument specifies the character position, not a byte position. Character comparisons are case sensitive and are determined by sort weights in the collation table in the database.

Example

The following example shows one way to use the PREFIX function:

```
create table prefix_table
     (
     colstring varchar(20),
     colchar char(1)
     );
 insert into prefix_table values ('string.with.dots', '.');
 insert into prefix_table values ('string-with-dashes', '-');
 select colstring, colchar, prefix(colstring, 1, '.') from prefix_table;
COLSTRING            COLCHAR              prefix(COLSTRING,1,.)
-------------------- -------------------- ---------------------
string.with.dots     .                    string
string-with-dashes   -                    string-with-dashes
  select colstring, colchar, prefix(colstring, 1, colchar) from prefix_table;
 COLSTRING            COLCHAR              prefix(COLSTRING,1,COLCHAR)
-------------------- -------------------- ---------------------------
string.with.dots     .                    string
string-with-dashes   -                    string
  select colstring, colchar, prefix(colstring, 1, 'X') from prefix_table;
 COLSTRING            COLCHAR              prefix(COLSTRING,1,X)
-------------------- -------------------- ---------------------
string.with.dots     .                    string.with.dots
string-with-dashes   -                    string-with-dashes
```

Compatibility

Progress extension

PRO_ARR_DESCAPE function

Removes escape characters from a single element of a character array. PRO_ARR_DESCAPE scans the *char_element* looking for the separator character (;) or an escape character (~). The function removes an escape character when it finds any of these constructs:

- Escape character followed by a separator character (~;)
- Escape character followed by another escape character (~~)

- Escape character followed by a NULL terminator (~\0)

Syntax

```
PRO_ARR_DESCAPE( 'char_element' ) ;
```

Parameter

char_element

The character representation of an array element, without any leading or trailing separators. Must be data type NVARCHAR, VARCHAR, or CHAR.

Note

char_element should not be the name of an array column, since the column contains true separators that would be destroyed by this function.

Examples

The following example returns the string 'aa;aa':

```
PRO_ARR_DESCAPE('aa~;aa') ;
```

The following example returns the string 'aa~aa'. There is no change, since another special character does not follow the escape character:

```
PRO_ARR_DESCAPE('aa~aa') ;
```

This example returns the string 'aa~;aa':

```
PRO_ARR_DESCAPE('aa~;aa') ;
```

PRO_ARR_ESCAPE function

Adds required escape characters to a single element of a character array.

PRO_ARR_ESCAPE scans the *char_element* looking for the separator character (;) or an escape character (~). The function inserts an additional escape character when it finds any of these constructs:

- Escape character followed by a separator character (~;)
- Escape character followed by another escape character (~~)
- Escape character followed by a NULL terminator (~\0)

Syntax

```
PRO_ARR_ESCAPE( 'char_element' )  ;
```

Parameter

char_element

The character representation of an array element, without any leading or trailing separators. Must be data type NVARCHAR, or VARCHAR, or CHAR.

Notes

- *char_element* must be data type NVARCHAR, VARCHAR, or CHAR.
- *char_element* must not be the name of an array column, since the column contains true separators that would be destroyed by this function.

Examples

The following example returns the string 'aa~;aa':

```
PRO_ARR_ESCAPE('aa;aa') ;
```

The following example returns the string 'aa~aa'. There is no change, since another special character does not follow the escape character:

```
PRO_ARR_ESCAPE('aa~aa') ;
```

This example returns the string 'aa~~;aa':

```
PRO_ARR_ESCAPE('aa~;aa') ;
```

PRO_ELEMENT function

Extracts one or more elements from an array column and returns the NVARCHAR or VARCHAR string between the specified positions, including any internal separator characters and any internal escape characters.

Syntax

```
PRO_ELEMENT ( 'array_style_expression', start_position, end_position ) ;
```

Parameters

array_style_expression

A string of data type VARCHAR or CHAR, with a semicolon (;) separating each element of the array.

start_position

The numeric position in the string marking the beginning of the element PRO_ELEMENT is to extract.

end_position

The numeric position in the string marking the end of the element to be extracted.

Notes

- The *array_style_expression* must be data type NVARCHAR, VARCHAR, or CHAR.
- The returned string does not include the leading separator of the first element, or the trailing separator (;) of the last element.
- Even when extracting only one element, the escape characters are included in the result.
- You must invoke PRO_ARR_DESCAPE to remove any escape characters.
- If the expression evaluates to NULL, the result is NULL.
- When accessing a sequence of multiple array elements if all the chosen array elements are NULL, then, the value of PRO_ELEMENT( ) is NULL. As any expression involving NULL evaluates to NULL, you should be careful when using the PRO_ELEMENT in this way when all the referenced array elements may be NULL. As an example, for a column named "col1" is an array of five elements of Date datatype, where, col1[2]..col5] are NULL, then, the value of PRO_ELEMENT("col1", 2, 5) is NULL.
- When the Authorized Data Truncation (ADT) is set to ON or OUTPUT , using the PRO_ELEMENT for var array might return the error “invalid array element reference”. To understand this better, refer **Example 2** of this topic.
- If an application wants to obtain a sequence of '?' values for a sequence of array elements each equal to NULL, use an expression combining the individual array elements. For example:

```
PRO_ELEMENT("col1",2,2) || ';' || PRO_ELEMENT("col1",3,3) || ';' ||
PRO_ELEMENT("col1",4,4) || ';' || PRO_ELEMENT("col1",5,5)
```

- While invoking PRO_ELEMENT, if Start=End, and the argument is NULL the result is (that is, the referenced array element is NULL) the result is '?'. For example, in the following function call, if sales_history[2] is NULL, then the PRO_ELEMENT returns ‘?’.

```
PRO_ELEMENT(sales_history, 2, 2)
```

Example 1

The following example returns the string 'bb':

```
PRO_ELEMENT('aa;bb;cc', 2, 2) ;
```

The next example returns the string 'aa;bb':

```
PRO_ELEMENT('aa;bb;cc', 1, 2) ;
```

This example returns the string 'aa~;aa':

```
PRO_ELEMENT('aa~;aa;bb;cc', 1, 1) ;
```

Example 2

Consider a var array `( cVar VARCHAR(5) VARARRAY[2])` into which ABL users insert the data `('123456789012345';'1234567890')`.

Now, SQL users execute the SQL query `select pro_elemenet( cvar,2,2) from table` when the Authorized Data Truncation (ADT) is set to on/output. The expected output with the current SQL behavior is to throw the error : `invalid array element reference`.

The reason for this behavior is that in case of a var array, while converting it into to string format, the truncation should happen on the entire array rather than on each element. The VARARRAY data type allows the size of an individual element value to exceed its declared size as long as the total size of the array is smaller than the array's SQL width.

So, for a var array, each element will be read incrementally until the entire data for the element is read. While doing so, if the total array length is exhausted, further data will not be read and the string produced so far will be returned.

In the above example, the first element is read first even though the second element is required and the array length is exhausted. So, the returned string does not have a reference to second element and it returns `invalid array element reference`.

This is an expected behavior. However, to get the second element, SQL users can and should use the subscript notation: `my_array[2]`.

QUARTER

Returns the quarter in the year specified by the argument as a short integer value in the range of 1-4.

Syntax

```
QUARTER ( date_expression )
```

Notes

- The argument to the function must be of type `DATE`.
- If *`date_expression`* is supplied as a date literal, it can be any of the valid *`date_literal`* formats where the day specification (DD) precedes the month specification (MM).
- The result is of type `SHORT`.
- If the argument expression evaluates to `NULL`, the result is `NULL`.

Example

In this example, which illustrates the QUARTER function, the query requests all rows in the orders table where the order_date is in the third quarter of the year:

```
SELECT *
    FROM orders
    WHERE QUARTER (order_date) = 3 ;
```

Compatibility

ODBC compatible

RADIANS

Returns the number of radians in an angle specified in degrees by *expression*.

Syntax

```
RADIANS ( expression )
```

Notes

- *expression* specifies an angle in degrees.
- *expression* must evaluate to a numeric data type.

Example

This example illustrates the RADIANS function:

```
SELECT RADIANS(180) 'Radians in 180 degrees' FROM SYSPROGRESS.SYSCALCTABLE;
```

Compatibility

ODBC compatible

RAND

Returns a randomly generated number, using expression as an optional seed value.

Syntax

```
RAND ( [expression] )
```

Note

expression must be an INT (32-bit) data type.

Example

This example illustrates the RAND function, supplying an optional seed value of '3':

```
SELECT RAND(3) 'Random number using 3 as seed value'
     FROM MYMATH;
```

Compatibility

ODBC compatible

REPEAT

Returns a character string composed of *string_exp* repeated *count* times.

Syntax

```
REPEAT ( string_exp , count )
```

Notes

- The *string_exp* can be of the type fixed-length or variable-length CHARACTER.
- The count can be of type INTEGER, SMALLINT, or TINYINT.
- If any of the arguments of the expression evaluates to a NULL, the result is NULL.
- If the count is negative or zero, the result evaluates to NULL.
- *string_exp* and the result can contain multi-byte characters.

Example

The following example shows how to use the REPEAT function:

```
SELECT REPEAT(fld1,3) FROM test100WHERE fld1 = 'Afghanistan';
 REPEAT(FLD1,3)
-----------------
AfghanistanAfghanistanAfghanistan
 1 record selected
```

Compatibility

ODBC compatible

REPLACE

Replaces all occurrences of *string_exp2* in *string_exp1* with *string_exp3*.

Syntax

```
REPLACE ( string_exp1 , string_exp2 , string_exp3 )
```

Notes

- *string_exp* can be fixed-length or variable-length CHARACTER data types.
- If any of the arguments of the expression evaluates to NULL, the result is NULL.
- If the replacement string is not found in the search string, it returns the original string.
- Each occurrence of *string_exp* and the result can contain multi-byte characters. Character comparisons are case sensitive and are determined by sort weights in the collation table in the database.

Example

This example illustrates the REPLACE function, replacing the letters 'mi' in the last_name 'Smith' with the letters 'moo':

```
SELECT REPLACE ( last_name,'mi','moo' )
     FROM customer WHERE last_name = 'Smith';
 REPLACE(LAST_NAME,MI,MOO)
-------------------------
Smooth
 1 record selected
```

Compatibility

ODBC compatible

RIGHT

Returns the rightmost count of characters of string_exp.

Syntax

```
RIGHT ( string_exp , count )
```

Notes

- The *string_exp* can be fixed-length or variable-length CHARACTER data types.
- The *count* can be INTEGER, SMALLINT, or TINYINT data types.
- If any of the arguments of the expression evaluate to NULL, the result is NULL.

- If *count* is negative, the result evaluates to NULL.
- *string_exp* and the result can contain multi-byte characters. *count* represents the number of characters.

Example

This example illustrates the RIGHT function, selecting the rightmost six letters from the string 'Afghanistan':

```
SELECT RIGHT(fld1,6) FROM  test100 WHERE fld1 = 'Afghanistan';
 RIGHT(FLD1,6)
---------------
nistan
 1 record selected
```

Compatibility

ODBC compatible

ROUND

Returns the rounded value of a numeric expression.

Syntax

```
ROUND ( num_expression
[ , rounding_factor ]);
```

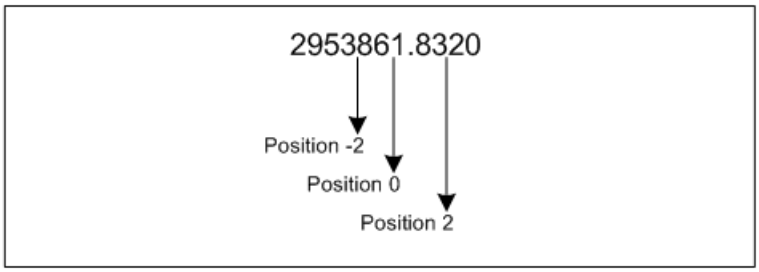
Notes

- *num_expression* must be numeric or must be convertible to numeric.
- *num_expression* must be one of these supported data types:
  - INTEGER
  - TINYINT
  - SMALLINT
  - NUMBER
  - FLOAT
  - DOUBLE PRECISION
- If the data type of *num_expression* is not a supported type, ROUND returns an error message.
- The *num_expression* is rounded to the next higher digit when:
  - The digit before a negative *rounding_factor* is 5 or greater
  - The digit after a positive *rounding_factor* is 5 or greater
- The *num_expression* is rounded to the next lower digit when:
  - The digit before a negative *rounding_factor* is 4 or less

- The digit after a positive *rounding_factor* is 4 or less

- *rounding_factor* is an integer between -32 and +32 inclusive, and indicates the digit position to which you want to round *num_expression*. The following figure illustrates how the digit positions are numbered. In the figure below, the *num_expression* is 2953861.8320.

  Figure 1: ROUND digit positions

  - If you do not specify a *rounding_factor*, the function rounds *num_expression* to digit 0 (the ones place).
  - To round to the right of the decimal point, specify a positive *rounding_factor*.
  - To round to the left of the decimal, specify a negative *rounding_factor*.

Example

This example illustrates four calls to the ROUND function:

```
-- rounding_factor 2 returns 2953861.83
ROUND ( 2953861.8320, 2 )
  -- rounding_factor -2 returns 2953900.00
ROUND ( 2953861.8320, -2 )
  -- rounding_factor 0 returns 2953862.00
ROUND ( 2953861.8320, 0 )
  -- No rounding_factor argument also returns 2953862.00
ROUND ( 2953861.8320 )
```

In each case the *num_expression* is 2953861.8320. In the first call the *rounding_factor* is 2, in the second call the *rounding_factor* is -2, in the third call the*rounding_factor* is 0, and in the fourth call no *rounding_factor* is specified.

Compatibility

Progress extension

ROWID

Returns the row identifier of the current row in a table. This function takes no arguments. The ROWID of a row is determined when the row is inserted into the table. Once assigned, the ROWID remains the same for the row until the row is deleted. At any given time, each row in a table is uniquely identified by its ROWID. Using its ROWID is the most efficient way of selecting the row.

Syntax

```
ROWID
```

Notes

The ROWID function returns a string of up to 19 characters in length.

Example

This example illustrates the ROWID function, returning all columns from the row in the customers table where the ROWID = '10':

```
SELECT *
    FROM customers
    WHERE ROWID = '10';
```

Compatibility

Progress extension

RPAD

Pads the character string corresponding to the first argument on the right with the character string corresponding to the third argument. After the padding, the length of the result is equal to the value of the second argument *length*.

Syntax

```
RPAD ( char_expression, length [ , pad_expression ] )
```

Example

This example illustrates two ways to use the RPAD function:

```
SELECT RPAD (last_name, 30)
     FROM customer ;
 SELECT RPAD (last_name, 30, '.')
     FROM customer ;
```

Notes

- The first argument to the function must be of type CHARACTER. The second argument to the function must be of type INTEGER. The third argument, if specified, must be of type CHARACTER. If the third argument is not specified, the default value is a string of length 1 containing one blank.
- If *L1* is the length of the first argument and L2 is the value of the second argument:
  - If *L1* is less than *L2*, the number of characters padded is equal to *L2* minus *L1*.

- If *L1* is equal to *L2*, no characters are padded and the result string is the same as the first argument.
- If *L1* is greater than *L2*, the result string is equal to the first argument truncated to the first *L2* characters.

- The result is of type CHARACTER.
- If the argument expression evaluates to NULL, the result is NULL.
- *char_expression* and *pad_expression*can contain multi-byte characters. *length*represents the number of characters in the result.

Compatibility

Progress extension

RTRIM

Removes all the trailing characters in *char_expression* that are present in *char_set* and returns the resultant string. The last character in the result is guaranteed not to be in *char_set*. If you do not specify a *char_set*, trailing blanks are removed.

Syntax

```
RTRIM ( char_expression[ , char_set] )
```

Notes

- The first argument to the function must be of type CHARACTER.
- The second argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to NULL, the result is NULL.
- The *char_expression*, the character set specified by *char_set*, and the result can contain multi-byte characters. Character comparisons are case sensitive and are determined by the collation table in the database.

Example

This example illustrates the RTRIM function:

```
SELECT RPAD ( RTRIM (addr, ' '), 30, '.')
     FROM customer ;
```

Compatibility

ODBC compatible

SECOND

Returns the seconds in the argument as a short integer value in the range of 0-59.

Syntax

```
SECOND ( time_expression )
```

Notes

- The argument to the function must be of type `TIME`.
- The argument must be specified in the format *`HH:MI:SS`*.
- The result is of type `SHORT`.
- If the argument expression evaluates to `NULL`, the result is `NULL`.

Example

This example illustrates the `SECOND` function, requesting all columns from rows in the arrivals table where the `in_time` column is less than or equal to '`40`':

```
SELECT * FROM arrivals WHERE SECOND (in_time) <= 40 ;
```

Compatibility

ODBC compatible

SIGN

Returns `1` if *`expression`* is positive, `-1` if *`expression`* is negative, or zero if *`expression`* is zero.

Syntax

```
SIGN ( expression )
```

Notes

`expression` must evaluate to a `NUMERIC` data type.

Example

This example illustrates the `SIGN` function:

```
SELECT SIGN(-14) 'Sign' FROM MYMATH;
```

Compatibility

ODBC compatible

SIN

Returns the sine of *expression*.

Syntax

```
SIN ( expression )
```

Notes

- SIN takes an angle (*expression*) and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the hypotenuse.
- *expression* specifies an angle in radians.
- *expression* must evaluate to an approximate numeric data type.
- To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.

Example

This example illustrates the SIN trigonometric function:

```
select sin(45 * pi()/180) 'Sine of 45 degrees' from MYMATH;
 SINE OF 45 DEGREES
------------------
0.707106781186547
 1 record selected
```

Compatibility

ODBC compatible

SQRT

Returns the square root of *expression*.

Syntax

```
SQRT ( expression )
```

Notes

- The value of *expression* must be positive.

- *expression* must evaluate to an approximate numeric data type.

Example

This example illustrates the SQRT function, requesting the square root of the value ‘28':

```
SELECT SQRT(28) 'square root of 28' FROM MYMATH;
```

Compatibility

ODBC compatible

SUBSTR

Returns the substring of the character string corresponding to the first argument starting at `start_pos` and `length` characters long. If the third argument `length` is not specified, the substring starting at `start_pos` up to the end of `char_expression` is returned.

Syntax

```
SUBSTR ( char_expression, start_pos [ , length ] )
```

Notes

- The first argument must be of type CHARACTER. It can be any meaningful character value (for example, a literal expression, database column, or parameter).
- The second argument must be of type INTEGER. It indicates the starting position from which the substring result is extracted.
- The third argument, if specified, must be of type INTEGER. It indicates the number of characters the substring function will extract.
- The values for specifying position in the character string start from 1. The first character in a string is at position 1, the second character is at position 2, and so on.
- The result is of type CHARACTER.
- If any of the argument expressions evaluate to NULL, the result is NULL.
- *char_expression* and the result can contain multi-byte characters.
- If the value of *start_pos* is:
  - Smaller than 0, the function returns a "Bad argument" error
  - Bigger than the actual length of the string value, the function returns an empty zero length substring
- If the value of *length* is:
  - Smaller than 0, the function returns a "Bad argument" error.
  - Bigger than the actual length of the substring (from *start_pos* to the end of the literal), the function returns the substring from *start_pos* to the end of the literal.

- Bigger than the actual length of the substring (from *start_pos* up to the end of the column's row data), the function returns the substring from the *start_pos* to the end of the column's row data. The function returns this, **even when the substring result exceeds the column's SQL width**.
- Bigger than 0 and the column's row data exceeds the column's SQL width, the function returns the substring.

- If *length* is not specified, the function returns the substring from *start_pos* to the end of the literal.
- The function returns the end of the column's row data **if the length of the substring is not bigger than the column's SQL width**.Otherwise, the function returns the substring truncated to the column's SQL width.

Example

This example illustrates the SUBSTR function:

```
SELECT last_name, '(', SUBSTR (phone, 1, 3) , ')',
          SUBSTR (phone, 4, 3), '-',
          SUBSTR (phone, 7, 4)
  FROM customer ;
```

Compatibility

Progress extension

SUBSTRING (ODBC compatible)

Returns the substring of the character string corresponding to the first argument starting at start_pos and length characters long. If the third argument *length* is not specified, the substring starting at start_pos up to the end of char_expression is returned.

Syntax

```
SUBSTRING ( char_expression, start_pos
  [ , length ] )
```

Notes

- The first argument must be of type CHARACTER. It can be any meaningful character value (for example, a literal expression, database column, or parameter).
- The second argument must be of type INTEGER. It indicates the starting position from which the substring result is extracted.
- The third argument, if specified, must be of type INTEGER. It indicates the number of characters the substring function will extract.
- The values for specifying position in the character string start from 1. The first character in a string is at position 1, the second character is at position 2, and so on.
- The result is of type CHARACTER.
- If any of the argument expressions evaluate to NULL, the result is NULL.
- *char_expression* and the result can contain multi-byte characters.

- If the value of *start_pos* is:
  - Smaller than 0, the function returns a "Bad argument" error
  - Bigger than the actual length of the string value, the function returns an empty zero length substring
- If the value of *length* is:
  - Smaller than 0, the function returns a "Bad argument" error.
  - Bigger than the actual length of the substring (from *start_pos* to the end of the literal), the function returns the substring from *start_pos* to the end of the literal.
  - Bigger than the actual length of the substring (from *start_pos* up to the end of the column's row data), the function returns the substring from the *start_pos* to the end of the column's row data. The function returns this, **even when the substring result exceeds the column's SQL width**.
  - Bigger than 0 and the column's row data exceeds the column's SQL width, the function returns the substring.
- If *length* is not specified, the function returns the substring from *start_pos* to the end of the literal.
- The function returns the end of the column's row data **if the length of the substring is not bigger than the column's SQL width**. Otherwise, the function returns the substring truncated to the column's SQL width.

Example

This example illustrates the SUBSTRING function:

```
SELECT last_name, '(', SUBSTRING (phone, 1, 3) , ')',
          SUBSTRING (phone, 4, 3), '-',
          SUBSTRING (phone, 7, 4)
  FROM customer ;
```

Compatibility

ODBC compatible

SUFFIX

Returns the substring of a character string starting after the position specified by start_pos and the second char_expression, to the end of the string.

Syntax

```
SUFFIX (char_expression , start_pos , char_expression )
```

Parameters

char_expression

Evaluates to a character string, typically a character-string literal or column name. If the expression evaluates to NULL, SUFFIX returns NULL.

start_pos

Evaluates to an integer value. SUFFIX searches the string specified in the first argument starting at that position. A value of 1 indicates the first character of the string.

char_expression

Evaluates to a single character. SUFFIX returns the substring that begins with that character. If SUFFIX does not find the character after *start_pos*, it returns NULL. If the expression evaluates to more than one character, SUFFIX ignores all but the first character.

Notes

Each *char_expression* and the result can contain multi-byte characters. The *start_pos* argument specifies the character position, not a byte position. Character comparisons are case sensitive and are determined by sort weights in the collation table in the database.

Example

This example illustrates two ways to use the SUFFIX function:

```
SELECT C1, C2, SUFFIX(C1, 6, '.') FROM T1;
C1      C2 SUFFIX(C1,6,.
--      -- -------------
test.pref  .
pref.test  s
 2 records selected
  SELECT C1, C2, SUFFIX(C1, 1, C2) FROM T1;
 C1      C2 SUFFIX(C1,1,C
--      -- -------------
test.pref  . pref
pref.test  s  t
 2 records selected
```

Compatibility

Progress extension

SUM

Returns the sum of the values in a group. The keyword DISTINCT specifies that the duplicate values are to be eliminated before computing the sum.

Syntax

```
SUM ( {[ALL]expression}|{ DISTINCT column_ref} )
```

Notes

- The argument *column_ref* or *expression* can be of any type.
- The result of the function is of the same data type as that of the argument except that the result is of type INTEGER when the argument is of type SMALLINT or TINYINT.

- The result can have a NULL value.

Example

This example illustrates the SUM function:

```
SELECT SUM (amount)
     FROM orders
     WHERE order_date = SYSDATE ;
```

SYSDATE

Returns the current date as a DATE value. This function takes no arguments, and the trailing parentheses are optional.

Syntax

```
SYSDATE [ ( ) ]
```

Example

This example illustrates the SYSDATE function, inserting a new row into the objects table, setting the create_date column to the value of the current date:

```
INSERT INTO objects (object_owner, object_id, create_date)
     VALUES (USER, 1001, SYSDATE) ;
```

Compatibility

Progress extension

SYSTIME

Returns the current time as a TIME value to the nearest second. This function takes no arguments, and the trailing parentheses are optional. SQL statements can refer to SYSTIME anywhere they can refer to a TIME expression.

Syntax

```
SYSTIME [ ( ) ]
```

Example

This example illustrates the SYSTIME function, inserting a new row into the objects table, setting the create_time column to the value of the current time:

```
INSERT INTO objects (object_owner, object_id, create_time)
     VALUES (USER, 1001, SYSTIME) ;
```

Compatibility

Progress extension

SYSTIMESTAMP

Returns the current date and time as a TIMESTAMP value. This function takes no arguments, and the trailing parentheses are optional.

Syntax

```
SYSTIMESTAMP [ ( ) ]
```

Example

This example illustrates different formats for SYSDATE, SYSTIME, and SYSTIMESTAMP:

```
SELECT SYSDATE FROM test;
 SYSDATE
-------
09/13/2003
 1 record selected
  SELECT SYSTIME FROM test;
 SYSTIME
-------
14:44:07:000
 1 record selected
  SELECT SYSTIMESTAMP FROM test;
 SYSTIMESTAMP
------------
2003-09-13 14:44:15:000
 1 record selected
```

Compatibility

Progress extension

TAN

Returns the tangent of *expression*.

Syntax

```
TAN ( expression )
```

Notes

- TAN takes an angle (*expression*) and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.
- *expression* specifies an angle in radians.
- *expression* must evaluate to an approximate numeric data type.
- To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.

Example

The following example shows how to use the TAN function:

```
select tan(45 * pi()/180) 'Tangent of 45 degrees' from
     MYMATH;
 TANGENT OF 45 DEGREES
----------------------
1.000000000000000
 1 record selected
```

Compatibility

ODBC compatible

TO_CHAR

Converts the given expression to character form and returns the result. The primary use for TO_CHAR is to format the output of date-time expressions through the *format_string* argument.

Syntax

```
TO_CHAR ( expresion [ , format_string ] )
```

Parameters

expression

Converts to character form. It must evaluate to a value of the date or time data type to use the format_string.

format_string

Specifies the format of the output. SQL ignores the format string if the *expression* argument does not evaluate to a date or time.

Notes

- The first argument to the function can be of any type.
- The second argument, if specified, must be of type CHARACTER.
- The result is of type CHARACTER.
- The *format* argument can be used only when the type of the first argument is DATE.
- If any of the argument expressions evaluates to NULL, the result is NULL.

Compatibility

Progress extension

TO_DATE

Converts the given date literal to a date value.

Syntax

```
TO_DATE ( date_literal )
```

Notes

- The result is of type DATE.
- Supply the date literal in any valid format.

Example

This example illustrates the TO_DATE function, returning all columns from rows in the orders table where the order_date column is earlier or equal to the date '12/31/2003':

```
SELECT *
    FROM orders
    WHERE order_date <= TO_DATE ('12/31/2003') ;
```

Compatibility

Progress extension

TO_NUMBER

Converts the given character expression to a number value.

Syntax

```
TO_NUMBER ( char_expression )
```

Notes

- The argument to the function must be of type CHARACTER.
- The result is of type NUMERIC.
- If any of the argument expressions evaluates to NULL, the result is NULL.

Example

This example illustrates the TO_NUMBER function and the SUBSTR function:

```
Select*
             From pub.customer
             WHERE (CASE SUBSTR(phone,1,1))
                     WHEN '('THEN SUBSTR(phone,2,3)
                     ELSE    substr(PHONE,1,3) ) =603;
```

Compatibility

Progress extension

TO_TIME

Converts the given time literal to a time value.

Syntax

```
TO_TIME ( time_literal )
```

Notes

- The result is of type TIME.
- Supply the time literal in any valid format.

Example

The following example shows how to use the TO_DATE and the TO_TIME functions:

```
SELECT * FROM orders
    WHERE order_date < TO_DATE ('05/15/2003')
    AND order_time < TO_TIME ('12:00:00') ;
```

Compatibility

Progress extension

TO_TIMESTAMP

Converts the given timestamp literal to a timestamp value.

Syntax

```
TO_TIMESTAMP ( timestamp_lit )
```

Notes

- The result is of type TIME.
- Supply the timestamp literal in any valid format.

Example

The following example shows how to use the TO_TIMESTAMP function:

```
SELECT * FROM DTEST WHERE C3 = TO_TIMESTAMP('4/18/03 10:41:19')
```

Compatibility

Progress Extension

TRANSLATE

Translates each character in *char_expression* that is in *from_set* to the corresponding character in *to_set*. The translated character string is returned as the result.

Syntax

```
TRANSLATE ( char_expression , from_set , to_set )
```

Notes

- *char_expression*, *from_set*, and *to_set* can be any character expression.
- For each character in *char_expression*, TRANSLATE checks for the same character in *from_set*.
- If it is in *from_set*, TRANSLATE translates it to the corresponding character in *to_set* (if the character is the*n*th character in *from_set*, the nth character in *to_set*).
- If the character is not in *from_set* TRANSLATE does not change it.
- If *from_set* is longer than *to_set*, TRANSLATE does not change trailing characters in *from_set* that do not have a corresponding character in *to_set*.
- If either *from_set* or *to_set* is NULL, TRANSLATE does nothing.

Example

This example substitutes underscores for spaces in customer names:

```
SELECT TRANSLATE (customer_name, ' ', '_')
   "TRANSLATE Example" from customers;
 TRANSLATE EXAMPLE
-----------------
Sports_Cars_Inc._____________________________
Mighty_Bulldozer_Inc._____________________________
Ship_Shapers_Inc._____________________________
Tower_Construction_Inc._____________________________
Chemical_Construction_Inc._____________________________
Aerospace_Enterprises_Inc._____________________________
Medical_Enterprises_Inc._____________________________
Rail_Builders_Inc._____________________________
Luxury_Cars_Inc._____________________________
Office_Furniture_Inc._____________________________
 10 records selected
```

Compatibility

Progress extension

UCASE

Returns the result of the argument character expression after converting all the characters to uppercase. UCASE is identical to UPPER, but provides ODBC-compatible syntax.

Syntax

```
UCASE ( char_expression )
```

Notes

- The argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to NULL, the result is NULL.
- A *char_expression* and the result can contain multi-byte characters. The uppercase conversion is determined by the case table in the *convmap* file. The default case table is BASIC.

Example

This example illustrates the UCASE function, returning columns from rows in the customer table where the last_name column, after being converted to uppercase, is equal to 'SMITH':

```
SELECT *
    FROM customer
    WHERE UCASE (last_name) = 'SMITH' ;
```

Compatibility

ODBC compatible

UPPER

Returns the result of the argument character expression after converting all the characters to uppercase.

Syntax

```
UPPER ( char_expression )
```

Notes

- The argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to NULL, the result is NULL.
- A *char_expression* and the result can contain multi-byte characters. The uppercase conversion is determined by the case table in the convmap file. The default case table is BASIC.

Example

This example illustrates the UPPER function, returning columns from rows in the customer table where the last_name column, after being converted to uppercase, is equal to 'SMITH':

```
SELECT *
    FROM customer
    WHERE UPPER (last_name) = 'SMITH' ;
```

Compatibility

SQL compatible

USER

Returns a character-string identifier for the user of the current transaction, as determined by the host operating system. This function takes no arguments, and the trailing parentheses are optional.

Syntax

```
USER [ ( ) ]
```

Notes

SQL statements can refer to USER anywhere they can refer to a character string expression.

Compatibility

ODBC compatible

WEEK

Returns the week of the year as a short integer value in the range of 1-53.

Syntax

```
WEEK ( time_expression )
```

Notes

- The argument to the function must be of type DATE.
- If *date_expression* is supplied as a date literal, it can be any of the valid *date_literal* formats where the day specification (DD) precedes the month specification (MM).
- The result is of type SHORT.
- If the argument expression evaluates to NULL, the result is NULL.

Example

The query returns all columns from rows in the orders table where the order_date is in the fifth week of the year. This example illustrates the WEEK function:

```
SELECT *
    FROM orders
    WHERE WEEK (order_date) = 5 ;
```

Compatibility

ODBC compatible

YEAR

Returns the year as a short integer value in the range of 0-9999.

Syntax

```
YEAR ( date_expression )
```

Notes

- The argument to the function must be of type DATE.
- If *date_expression* is supplied as a date literal, it can be any of the valid *date_literal* formats where the day specification (*DD*) precedes the month specification (*MM*).

- The result is of type SHORT.
- If the argument expression evaluates to NULL, the result is NULL.

Example

The query returns all columns in rows in the orders table where the year in the order_date column is equal to ‘2003'. This example illustrates the YEAR function:

```
SELECT *
    FROM orders
    WHERE YEAR (order_date) = 2003;
```

Compatibility

ODBC compatible

3

OpenEdge SQL reserved words

This topic provides a list of words that have special syntactic meaning to SQL and cannot be used as identifiers for constants, variables, cursors, types, tables, records, subprograms, or packages.

Reserved words are keywords. You can use keywords as identifiers in SQL statements only if you delimit them with double quotation marks. If you use keywords without delimiting them, the statement generates one of the following errors:

```
error(-20003): Syntax error
error(-20049): Keyword used for a name
```

The following table provides a list of OpenEdge SQL reserved words:

Table 1: OpenEdge SQL reserved words

ABS	ACOS	ADD	ADD_MONTHS
AFTER	ALL	ALTER	AND
ANY	ANY_USER	APPLICATION_CONTEXT	AREA
ARRAY	AS	ASC	ASCII
ASIN	ATAN	ATAN2	AUDIT
AUDIT_ADMIN	AUDIT_ARCHIVE	AUDIT_INSERT	AUDIT_READ
AVG	BEFORE	BETWEEN	BIGINT

BINARY	BIT	BLOB	BY
CALL	CARD_CHANGES	CASCADE	CASE
CEILING	CHAR	CHARACTER	CHARACTER_LENGTH
CHARTOROWID	CHAR_LENGTH	CHECK	CHR
CLOB	CLOSE	CLUSTERED	COALESCE
COLGROUP	COLLATE	COLUMN	COMMIT
COMPRESS	CONCAT	CONNECT	CONSTRAINT
CONTAINS	CONVERT	COS	COUNT
CREATE	CROSS	CURDATE	CURRENT
CURRENT_USER	CURRVAL	CURSOR	CURTIME
CYCLE	DATABASE	DATAPAGES	DATE
DAYNAME	DAYOFMONTH	DAYOFWEEK	DAYOFYEAR
DBA	DB_NAME	DEC	DECIMAL
DECLARE	DECODE	DEFAULT	DEFINITION
DEGREES	DELETE	DESC	DIFFERENCE
DISCONNECT	DISTINCT	DOMAIN	DOUBLE
DROP	EACH	Z	Z
ELSE	END	ESCAPE	EVENT_GROUP
ZONE	EVERY	LOB	MULTI_TENANT
EVENT_GROUP	EXCLUSIVE	EXECUTE	EXISTS
EXP	EXTENT	FETCH	FILE
FLOAT	FLOOR	FOR	FOREIGN
FROM	FULL	GLOBAL	GRANT
GREATEST	HASH	HAVING	HOUR
IDENTIFIED	IFNULL	IN	INDEX
INDEXPAGES	INDICATOR	INITCAP	INNER
INOUT	INSERT	INSTR	INT

INTEGER	INTERSECT	INTO	IS
ISOLATION	JOIN	KEY	LAST_DAY
LCASE	LEAST	LEFT	LENGTH
LEVEL	LIKE	LINK	LIST
LOCATE	LOCK	LOG	LOG10
LONG	LOWER	LPAD	LTRIM
LVARBINARY	LVARCHAR	MAX	MAXVALUE
METADATA_ONLY	MIN	MINUS	MINUTE
MINVALUE	MOD	MODE	MODIFY
MONTH	MONTHNAME	MONTHS_BETWEEN	NATIONAL
NATURAL	NCHAR	NEWROW	NEXTVAL
NEXT_DAY	NOCOMPRESS	NOCYCLE	NOEXECUTE
NOLOCK	NOMAXVALUE	NOMINVALUE	NOT
NOW	NOWAIT	NULL	NULLIF
NULLVALUE	NUMBER	NUMERIC	NVL
OBJECT_ID	ODBCINFO	ODBC_CONVERT	OF
OFF	OLDROW	ON	OPEN
OPTION	OR	OUT	OUTER
PARTITION	PCTFREE	PI	POWER
OFFSET	POLICY	PRO_CATEGORY	PRO_PARTN_ROWID
PRECISION	PREFIX	PRIMARY	PRIVILEGES
PRO_ARRAY_ELEMENT	PRO_ARR_DESCAPE	PRO_ARR_ESCAPE	PRO_ASSIGN
PRO_CAN_CREATE	PRO_CAN_DELETE	PRO_CAN_DUMP	PRO_CAN_LOAD
PRO_CAN_READ	PRO_CAN_WRITE	PRO_CASE_SENSITIVE	PRO_CLIENT_FIELD_TRIGGER
PRO_CLIENT_FILE_TRIGGER	PRO_COL_LABEL	PRO_CONNECT	PRO_CRC
PRO_CREATE	PRO_DATA_TYPE	PRO_DEFAULT_INDEX	PRO_DELETE
PRO_DESCRIPTION	PRO_DUMP_NAME	PRO_ELEMENT	PRO_ENABLE_64BIT_SEQUENCES

PRO_ENABLE_LARGE_KEYS	PRO_FIND	PRO_FORMAT	PRO_FROZEN
PRO_HELP	PRO_HIDDEN	PRO_LABEL	PRO_LOB_SIZE_TEXT
PRO_AREA_NAME	PRO_INITIAL	PROCEDURE	PRODEFAULT
PRO_NAME	PRO_ODBC	PRO_ORDER	PRO_OVERRIDEABLE
PRO_PROCNAME	PRO_REPL_CREATE	PRO_REPL_DELETE	PRO_REPL_WRITE
PRO_SA_COL_LABEL	PRO_SA_FORMAT	PRO_SA_HELP	PRO_SA_INITIAL
PRO_SA_LABEL	PRO_SA_VALMSG	PRO_SERVER	PRO_SQL_WIDTH
PRO_STATUS	PRO_TYPE	PRO_UNIFIED_SCHEMA	PRO_VALEXP
PRO_VALMSG	PRO_VIEW_AS	PRO_WORD	PRO_WRITE
PUBLIC	QUARTER	QUERY_TIMEOUT	RADIANS
RAND	RANGE	READPAST	REAL
RECID	REFERENCES	REFERENCING	RENAME
REPEAT	REPLACE	RESOURCE	RESTRICT
RESULT	RETURN	REVOKE	RIGHT
ROLLBACK	ROUND	ROW	ROWID
ROWIDTOCHAR	ROWNUM	RPAD	RTRIM
ROWCOUNT	ROWS	STDDEV	STDDEV_POP
SCHEMA	SEARCHED_CASE	SECOND	SELECT
SEQUENCE	SEQUENCE_CURRENT	SEQUENCE_NEXT	SET
SHARE	SHOW	SIGN	SIMPLE_CASE
SIN	SIZE	SMALLINT	SOME
SOUNDEX	SPACE	SQL_BIGINT	SQL_BINARY
SQL_BIT	SQL_CHAR	SQL_DATE	SQL_DECIMAL
SQL_DOUBLE	SQL_FLOAT	SQL_INTEGER	SQL_LONGVARBINARY
SQL_LONGVARCHAR	SQL_NUMERIC	SQL_REAL	SQL_SMALLINT
SQL_TIME	SQL_TIMESTAMP	SQL_TINYINT	SQL_TSI_DAY
SQL_TSI_FRAC_SECOND	SQL_TSI_HOUR	SQL_TSI_MINUTE	SQL_TSI_MONTH

SQL_TSI_QUARTER	SQL_TSI_SECOND	SQL_TSI_WEEK	SQL_TSI_YEAR
SQL_VARBINARY	SQL_VARCHAR	SQRT	STATEMENT
STATISTICS	STDDEV	STDDEV_POP	STDDEV_SAMP VARIANCE
STORAGE_ATTRIBUTES	STORAGE_MANAGER	STORE_IN_SQLENG	SUBPARTITION
SUBSTR	SUBSTRING	SUFFIX	SUM
STDDEV_SAMP	SUPER_TENANT	SYSTIMESTAMP_UTC	TASK_AGENT
SUSER_NAME	SYNONYM	SYSDATE	SYSTIME
SYSTIMESTAMP	SYSTIMESTAMP_TZ	SYSTIMESTAMP_UTC	TABLE
TENANTID_TBL	TENANTNAME_TBL	THRESHOLD_MON	THROTTLE_SERVICE
TAN	THEN	TIME	TIMESTAMPADD
TIMESTAMPDIFF	TINYINT	TO	TOP
TO_CHAR	TO_DATE	TO_NUMBER	TO_TIME
TO_TIMESTAMP	TO_TIMESTAMP_TZ	TRANSACTION	TRANSLATE
TRIGGER	TYPE	UCASE	UID
UNION	UNIQUE	UPDATE	UPPER
USER	USER_ID	USER_NAME	USING
VALUES	VARCHAR	VAR_SAMP	VARYING
UPDATES	VARIANCE	VAR_POP	VAR_SAMP
VIEW	WAIT	WEEK	WHEN
WHERE	WITH	WORK	YEAR
CAST			

4

OpenEdge SQL error messages

This topic provides information on error messages generated by the various components of OpenEdge SQL. Error message information includes:

- Error code
- SQLSTATE value
- Class condition
- Subclass message

For details, see the following topics:

- Overview
- Error codes, SQLSTATE values, and messages

Overview

In addition to the OpenEdge-specific error codes, error conditions have an associated SQLSTATE value. SQLSTATE is a five-character status parameter whose value indicates the condition status returned by the most recent SQL statement. The first two characters of the SQLSTATE value specify the class code and the last three characters specify the subclass code:

- Class codes of a-h and 0-4 are reserved by the SQL standard. For those class codes only, subclass codes of a-h and 0-4 are also reserved by the standard.
- Subclasses S and T and class IM are reserved by the ODBC standard.

- Class codes of i-z and 5-9 are specific to database implementations such as OpenEdge SQL. All subclass codes in those classes are implementation defined except as noted for ODBC.

Error codes, SQLSTATE values, and messages

The following table is a list of OpenEdge SQL error messages, ordered by error code number. The table shows the corresponding SQLSTATE value for each message.

Table 2: OpenEdge SQL error codes and messages

Error code	SQLSTATE value	Class condition	Subclass message
00000	00000	Successful completion	***status okay.
100L	02000	No data	**sql not found.
10002	22503	Data exception	Tuple not found for the Specified TID.
10012	N0N12	Flag	ETPL_SCAN_EOP.
10013	22914	Data Exception	No more records to be fetched.
10100	2150b	Cardinality violation	Too many fields exist.
10101	70701	OpenEdge/SQL MM error	No more records exist.
10102	2350i	Integrity constraint	Duplicate primary/index key value.
10104	M0M06	OpenEdge/SQL rss error	Specified index method is not supported.
10107	N0N07	Flag	EIX_SCAN_EOP flag is set.
10108	50903	OpenEdge/SQL rds error	Duplicate record specified.
10301	M0901	OpenEdge/SQL rss error	Table is locked and LCK_NOWAIT.
10400	22501	Data exception	Invalid file size for alter log statement.
10920	22521	Data exception	Already existing value specified.
11100	50901	OpenEdge/SQL rds error	Invalid transaction id.
11102	50903	OpenEdge/SQL rds error	TDS area specified is not found.
11103	50504	OpenEdge/SQL rds error	TDS not found for binding.
11104	50505	OpenEdge/SQL rds error	Transaction aborted.
11105	50506	OpenEdge/SQL rds error	Transaction error.
11109	50510	OpenEdge/SQL rds error	Invalid transaction handle.

Error code	SQLSTATE value	Class condition	Subclass message
11111	50912	OpenEdge/SQL rds error	Invalid isolation level.
11300	M0M00	OpenEdge/SQL rss error	Specified INFO type is not supported.
11301	M0M01	OpenEdge/SQL rss error	Specified index type is not supported.
16001	22701	Data exception	MM- No data block.
16002	70702	OpenEdge SQLMM error	MM- Bad swap block.
16003	70703	OpenEdge SQLMM error	MM- No cache block.
16004	22704	Data exception	MM- Invalid row number.
16005	70705	OpenEdge SQL MM error	MM- Invalid cache block.
16006	70706	OpenEdge SQL MM error	MM- Bad swap file.
16007	70707	OpenEdge SQL MM error	MM- Row too big.
16008	70708	OpenEdge SQL MM error	MM- Array initialized.
16009	70709	OpenEdge SQL MM error	MM- Invalid chunk number.
16010	70710	OpenEdge SQL MM error	MM- Cannot create table.
16011	70711	OpenEdge SQL MM error	MM- Cannot alter table.
16012	70712	OpenEdge SQL MM error	MM- Cannot drop table.
16020	70713	OpenEdge SQL MM error	MM- TPL ctor error.
16021	70714	OpenEdge SQL MM error	MM- Insertion error.
16022	70715	OpenEdge SQL MM error	MM- Deletion error.
16023	70716	OpenEdge SQL MM error	MM- Updation error.
16024	70717	OpenEdge SQL MM error	MM- Fetching error.
16025	70718	OpenEdge SQL MM error	MM- Sorting error.
16026	70719	OpenEdge SQL MM error	MM- Printing error.
16027	70720	OpenEdge SQL MM error	MM- TPLSCAN ctor error.
16028	70721	OpenEdge SQL MM error	MM- Scan fetching error.
16030	70722	OpenEdge SQL MM error	MM- Can't create index.

Error code	SQLSTATE value	Class condition	Subclass message
16031	70723	OpenEdge SQL MM error	MM- Can't drop index.
16032	70724	OpenEdge SQL MM error	MM- IXSCAN ctor error.
16033	70725	OpenEdge SQL MM error	MM- IX ctor error.
16034	70726	OpenEdge SQL MM error	MM- IX deletion error.
16035	70727	OpenEdge SQL MM error	MM- IX appending error.
16036	70728	OpenEdge SQL MM error	MM- IX insertion error.
16037	70729	OpenEdge SQL MM error	MM- IX scan fetching error.
16040	70730	OpenEdge SQL MM error	MM- Begin transaction.
16041	70731	OpenEdge SQL MM error	MM- Commit transaction.
16042	40000	Transaction rollback	***MM- Rollback transaction.
16043	70732	OpenEdge SQL MM error	MM- Mark point.
16044	70733	OpenEdge SQL MM error	MM- Rollback savepoint.
16045	70734	OpenEdge SQL MM error	MM- Set & Get isolation.
16050	70735	OpenEdge SQL MM error	MM- TID to char.
16051	70736	OpenEdge SQL MM error	MM- Char to TID.
16054	70737	OpenEdge SQL MM error	MM- Bad value list size to indirect sort.
20000	50501	OpenEdge SQL rds error	SQL internal error.
20001	50502	OpenEdge SQL rds error	Memory allocation failure.
20002	50503	OpenEdge SQL rds error	Open database failed.
20003	2a504	Syntax error	Syntax error.
20004	28505	Invalid auth specs	User not found.
20005	22506	Data exception	Table/View/Synonym not found.
20006	22507	Data exception	Column not found/specified.
20007	22508	Data exception	No columns in table.
20008	22509	Data exception	Inconsistent types.

Error code	SQLSTATE value	Class condition	Subclass message
20009	22510	Data exception	Column ambiguously specified.
20010	22511	Data exception	Duplicate column specification.
20011	22512	Data exception	Invalid length.
20012	22513	Data exception	Invalid precision.
20013	22514	Data exception	Invalid scale.
20014	22515	Data exception	Missing input parameters.
20015	22516	Data exception	Subquery returns multiple rows.
20016	22517	Data exception	Null value supplied for a mandatory (not null) column.
20017	22518	Data exception	Too many values specified.
20018	22519	Data exception	Too few values specified.
20019	50520	OpenEdge SQL rds error	Cannot modify table referred to in subquery.
20020	42521	Access rule violation	Bad column specification for group by clause.
20021	42522	Access rule violation	Non-group-by expression in having clause.
20022	42523	Access rule violation	Non-group-by expression in select clause.
20023	42524	Access rule violation	Aggregate function not allowed here.
20024	0a000	Feature not supported	Sorry, operation not yet implemented.
20025	42526	Access rule violation	Aggregate functions nested.
20026	50527	OpenEdge SQL rds error	Too many table references.
20027	42528	Access rule violation	Bad field specification in order by clause.
20028	50529	OpenEdge SQL rds error	An index with the same name already exists.
20029	50530	OpenEdge SQL rds error	Index referenced not found.
20030	22531	Data exception	Table space with same name already exists.
20031	50532	OpenEdge SQL rds error	Cluster with same name already exists.
20032	50533	OpenEdge SQL rds error	No cluster with this name.

Error code	SQLSTATE value	Class condition	Subclass message
20033	22534	Data exception	Table space not found.
20034	50535	OpenEdge SQL rds error	Bad free <*specification_name*> specification.
20035	50536	OpenEdge SQL rds error	At least column spec or null clause should be specified.
20036	07537	Dynamic sql error	Not prepared.
20037	24538	Invalid cursor state	Executing select statement.
20038	24539	Invalid cursor state	Cursor not closed.
20039	24540	Invalid cursor state	Open for nonselect statement.
20040	24541	Invalid cursor state	Cursor not opened.
20041	22542	Data exception	Table/View/Synonym already exists.
20042	2a543	Syntax error	Distinct specified more than once in query.
20043	50544	OpenEdge SQL rds error	Tuple size too high.
20044	50545	OpenEdge SQL rds error	Array size too high.
20045	08546	Connection exception	File does not exist or not accessible.
20046	50547	OpenEdge SQL rds error	Field value not null for some tuples.
20047	42548	Access rule violation	Granting to self not allowed.
20048	42549	Access rule violation	Revoking for self not allowed.
20049	22550	Data exception	Keyword used for a name.
20050	21551	Cardinality violation	Too many fields specified.
20051	21552	Cardinality violation	Too many indexes on this table.
20052	22553	Data exception	Overflow error.
20053	08554	Connection exception	Database not opened.
20054	08555	Connection exception	Database not specified or improperly specified.
20055	08556	Connection exception	Database not specified or database not started.

Error code	SQLSTATE value	Class condition	Subclass message
20056	28557	Invalid auth specs	No DBA access rights.
20057	28558	Invalid auth specs	No RESOURCE privileges.
20058	40559	Transaction rollback	Executing SQL statement for an aborted transaction.
20059	22560	Data exception	No files in the table space.
20060	22561	Data exception	Table not empty.
20061	22562	Data exception	Input parameter size too high.
20062	42563	Syntax error	Full pathname not specified.
20063	50564	OpenEdge SQL rds error	Duplicate file specification.
20064	08565	Connection exception	Invalid attach type.
20065	26000	Invalid SQL statement name	Invalid statement type.
20066	33567	Invalid SQL descriptor name	Invalid sqlda.
20067	08568	Connection exception	More than one database cannot be attached locally.
20068	42569	Syntax error	Bad arguments.
20069	33570	Invalid SQL descriptor name	SQLDA size not enough.
20070	33571	Invalid SQL descriptor name	SQLDA buffer length too high.
20071	42572	Access rule violation	Specified operation not allowed on the view.
20072	50573	OpenEdge SQL rds error	Server is not allocated.
20073	2a574	Access rule violation	View query specification for view too long.
20074	2a575	Access rule violation	View column list must be specified as expressions are given.
20075	21576	Cardinality violation	Number of columns in column list is less than in select list.
20076	21577	Cardinality violation	Number of columns in column list is more than in select list.
20077	42578	Access rule violation	Check option specified for noninsertable view.

Error code	SQLSTATE value	Class condition	Subclass message
20078	42579	Access rule violation	Given SQL statement is not allowed on the view.
20079	50580	OpenEdge SQL rds error	More tables cannot be created.
20080	44581	Check option violation	View check option violation.
20081	22582	Data exception	Number of expressions projected on either side of set-op do not match.
20082	42583	Access rule violation	Column names not allowed in order by clause for this statement.
20083	42584	Access rule violation	Outerjoin specified on a complex predicate.
20084	42585	Access rule violation	Outerjoin specified on a sub-query.
20085	42586	Access rule violation	Invalid Outerjoin specification.
20086	42587	Access rule violation	Duplicate table constraint specification.
20087	21588	Cardinality violation	Column count mismatch.
20088	28589	Invalid auth specs	Invalid user name.
20089	22590	Data exception	System date retrieval failed.
20090	42591	Access rule violation	Table column list must be specified as expressions are given.
20091	2a592	Access rule violation	Query statement too long.
20092	2d593	Invalid transaction termination	No tuples selected by the subquery for update.
20093	22594	Data exception	Synonym already exists.
20094	hz595	Remote database access	Database link with same name already exists.
20095	hz596	Remote database access	Database link not found.
20096	08597	Connection exception	Connect String not specified/incorrect.
20097	hz598	Remote database access	Specified operation not allowed on a remote table.
20098	22599	Data exception	More than one row selected by the query.
20099	24000	Invalid cursor state	Cursor not positioned on a valid row.

Error code	SQLSTATE value	Class condition	Subclass message
20100	4250a	Access rule violation	Subquery not allowed here.
20101	2350b	Integrity constraint	No references for the table.
20102	2350c	Integrity constraint	Primary/Candidate key column defined null.
20103	2350d	Integrity constraint	No matching key defined for the referenced table.
20104	2350e	Integrity constraint	Keys in reference constraint incompatible.
20105	5050f	OpenEdge SQL rds error	Not allowed in read only isolation level.
20106	2150g	Cardinality violation	Invalid ROWID.
20107	hz50h	Remote database access	Remote database not started.
20108	0850i	Connection exception	Remote Network Server not started.
20109	hz50j	Remote database access	Remote database name not valid.
20110	0850k	Connection exception	TCP/IP Remote HostName is unknown.
20114	33002	Invalid SQL descriptor name	Fetched Value NULL & indicator var not defined.
20115	5050l	OpenEdge SQL rds error	References to the table/record present.
20116	2350m	Integrity constraint	Constraint violation.
20117	2350n	Integrity constraint	Table definition not complete.
20118	4250o	Access rule violation	Duplicate constraint name.
20119	2350p	Integrity constraint	Constraint name not found.
20120	22000	Data exception	**Use of reserved word.
20121	5050q	OpenEdge SQL rds error	Permission denied.
20122	5050r	OpenEdge SQL rds error	Procedure not found.
20123	5050s	OpenEdge SQL rds error	Invalid arguments to procedure.
20124	5050t	OpenEdge SQL rds error	Query conditionally terminated.
20125	0750u	Dynamic sql-error	Number of open cursors exceeds limit.
20126	34000	Invalid cursor name	***Invalid cursor name.

Error code	SQLSTATE value	Class condition	Subclass message
20127	07001	Dynamic sql-error	Bad parameter specification for the statement.
20128	2250x	Data Exception	Numeric value out of range.
20129	2250y	Data Exception	Data truncated.
20132	5050u	OpenEdge SQL rds error	Revoke failed because of restrict.
20134	5050v	OpenEdge SQL rds error	Invalid long data type column references.
20135	5050x	OpenEdge SQL rds error	Contains operator is not supported in this context.
20135	m0m01	OpenEdge SQL diagnostics error	Diagnostics statement failed.
20136	5050z	OpenEdge SQL rds error	Contains operator is not supported for this datatype.
20137	50514	OpenEdge SQL rds error	Index is not defined or does not support CONTAINS.
20138	50513	OpenEdge SQL rds error	Index on long fields requires that it can push down only CONTAINS.
20140	50512	OpenEdge SQL rds error	Procedure already exists.
20141	85001	OpenEdge SQL Stored procedure Compilation	Error in stored procedure compilation.
20142	86001	OpenEdge SQL Stored procedure Execution	Error in Stored Procedure Execution.
20143	86002	OpenEdge SQL Stored procedure Execution	Too many recursions in call procedure.
20144	86003	OpenEdge SQL Stored procedure Execution	Null value fetched.
20145	86004	OpenEdge SQL Stored procedure Execution	Invalid field reference.
20146	86005	OpenEdge SQL Triggers	Trigger with this name already exists.
20147	86006	OpenEdge SQL Triggers	Trigger with this name does not exist.
20148	86007	OpenEdge SQL Triggers	Trigger Execution Failed.
20152	22001	Data exception	Character string is too long.

Error code	SQLSTATE value	Class condition	Subclass message
20170	22P0	Data exception	An invalid reference to a sequence was used.
20172	22565	Data exception	Sequence already exists in current schema.
20173	0ALLV	Feature not supported	LIKE predicate for long data type uses unsupported feature.
20174	86010	OpenEdge SQL Triggers	Invalid colnum number specified for Trigger OLDROW/NEWROW getValue/setValue method.
20175	86011	OpenEdge SQL Triggers	Incompatible data type specified for Trigger OLDROW/NEWROW getValue/setValue method.
20176	85001	OpenEdge SQL Stored proc/Trigger	IO error while compiling stored procedure/trigger.
20178	OaC01	Feature not supported	Cannot rename table/column with check constraint.
20180	22915	Data exception	Long data exceeds column width.
20181	22916	Data exception	Long data exceeds maximum size of data that can be selected.
20211	22800	Data exception	Remote procedure call error.
20212	08801	Connection exception	SQL client bind to daemon failed.
20213	08802	Connection exception	SQL client bind to SQL server failed.
20214	08803	Connection exception	SQL NETWORK service entry is not available.
20215	08804	Connection exception	Invalid TCP/IP hostname.
20216	hz805	Remote database access	Invalid remote database name.
20217	08806	Connection exception	Network error on server.
20218	08807	Connection exception	Invalid protocol.
20219	2e000	Invalid connection name	***Invalid connection name.
20220	08809	Connection exception	Duplicate connection name.
20221	08810	Connection exception	No active connection.

Error code	SQLSTATE value	Class condition	Subclass message
20222	08811	Connection exception	No environment defined database.
20223	08812	Connection exception	Multiple local connections.
20224	08813	Connection exception	Invalid protocol in connect_string.
20225	08814	Connection exception	Exceeding permissible number of connections.
20226	80815	OpenEdge SQL snw error	Bad database handle.
20227	08816	Connection exception	Invalid host name in connect string.
20228	28817	Invalid auth specs	Access denied (Authorization failed).
20229	22818	Data exception	Invalid date value.
20230	22819	Data exception	Invalid date string.
20231	22820	Data exception	Invalid number strings.
20232	22821	Data exception	Invalid number string.
20233	22822	Data exception	Invalid time value.
20234	22523	Data exception	Invalid time string.
20235	22007	Data exception	Invalid time stamp string.
20236	22012	Data exception	Division by zero attempted.
20238	22615	Data exception	Error in format type.
20239	2c000	Invalid character set name	Invalid character set name specified.
20240	5050y	OpenEdge SQL rds error	Invalid collation name specified.
20241	08815	Connection exception	Service in use.
20300	90901	DBS error	Column group column does not exist.
20301	90902	DBS error	Column group column already specified.
20302	90903	DBS error	Column group name already specified.
20303	90904	DBS error	Column groups have not covered all columns.
20304	90905	DBS error	Column groups are not implemented in Progress storage.

Error code	SQLSTATE value	Class condition	Subclass message
20489	22948	Data exception	This is a retired command and is no longer functional.
20490	22542	Data exception	Password length has exceeded the maximum allowed limit.
23000	22563	OpenEdge SQL Data exception	Table create returned invalid table id.
23001	22564	OpenEdge SQL Data exception	Index create returned invalid index id.
25128	j0j28	OpenEdge SQL odbc trans layer	Query terminated as max row limit exceeded for a remote table.
25131	j0j29	OpenEdge SQL odbc trans layer	Unable to read column info from remote table.
30001	5050w	OpenEdge SQL rds error	Query aborted on user request.
30002	k0k02	OpenEdge SQL network interface	Invalid network handle.
30003	k0k03	OpenEdge SQL network interface	Invalid sqlnetwork INTERFACE.
30004	k0k04	OpenEdge SQL network interface	Invalid sqlnetwork INTERFACE procedure.
30005	k0k05	OpenEdge SQL network interface	INTERFACE is already attached.
30006	k0k06	OpenEdge SQL network interface	INTERFACE entry not found.
30007	k0k07	OpenEdge SQL network interface	INTERFACE is already registered.
30008	k0k08	OpenEdge SQL network interface	Mismatch in pkt header size and total argument size.
30009	k0k09	OpenEdge SQL network interface	Invalid server id.
30010	k0k10	OpenEdge SQL network interface	Reply does not match the request.
30011	k0k02	OpenEdge SQL network interface	Memory allocation failure.
30031	k0k11	OpenEdge SQL network interface	Error in transmission of packet.
30032	k0k12	OpenEdge SQL network interface	Error in reception of packet.
30033	k0k13	OpenEdge SQL network interface	No packet received.
30034	k0k14	OpenEdge SQL network interface	Connection reset.
30051	k0k15	OpenEdge SQL network interface	Network handle is inprocess handle.
30061	k0k16	OpenEdge SQL network interface	Could not connect to sql network daemon.

Error code	SQLSTATE value	Class condition	Subclass message
30062	k0k17	OpenEdge SQL network interface	Error in number of arguments.
30063	k0k18	OpenEdge SQL network interface	Requested INTERFACE not registered.
30064	k0k19	OpenEdge SQL network interface	Invalid INTERFACE procedure id.
30065	k0k20	OpenEdge SQL network interface	Requested server executable not found.
30066	k0k21	OpenEdge SQL network interface	Invalid configuration information.
30067	k0k22	OpenEdge SQL network interface	INTERFACE not supported.
30091	k0k23	OpenEdge SQL network interface	Invalid service name.
30092	k0k24	OpenEdge SQL network interface	Invalid host.
30093	k0k25	OpenEdge SQL network interface	Error in tcp/ip accept call.
30094	k0k26	OpenEdge SQL network interface	Error in tcp/ip connect call.
30095	k0k27	OpenEdge SQL network interface	Error in tcp/ip bind call.
30096	k0k28	OpenEdge SQL network interface	Error in creating socket.
30097	k0k29	OpenEdge SQL network interface	Error in setting socket option.
30101	k0k30	OpenEdge SQL network interface	Interrupt occurred.
40001	L0L01	OpenEdge SQL env error	Error in reading configuration.
210001	08P00	Connection exception	Failure to acquire share schema lock during connect.
210002	08004	Connection exception	Failure in finding DLC environment variable.
210003	08004	Connection exception	DLC environment variable exceeds maximum size <*max_size*> -> <*DLC path*>.
210004	08004	Connection exception	Error opening convmap.cp file <*filename*> <*path*>.
210005	P1000	Unavailable resource	Failure getting lock table on table <*table_name*>.
210011	08004	Internal error	Fatal error identifying database log in SQL.
210012	22P00	Data exception	Column <*column_name*> in table <*table_name*> has value exceeding its max length or precision.

Error code	SQLSTATE value	Class condition	Subclass message
210013	08004	Connection exception	Unable to complete server connection. <*function_name*>; reason <*summary_of_reason*>.
210014	22P01	Data exception	Column values too big to make key. Table <*table_name*>; index <*index_name*>.
210015	P1000	Unavailable resource	Failure getting record lock on a record table <*table_name*>.
210016	P1001	Unavailable resource	Lock table is full.
210017	P1002	Unavailable resource	Failure to acquire exclusive schema lock for DDL operation.
210018	0AP01	Unsupported feature	Update of word indexes not yet supported. Table <*table_name*>, index <*index_name*>.
210019	0A000	Unsupported feature	Scan of word indexes not yet supported. Table <*table_name*>, index <*index_name*>.
210020	0AP03	Unsupported feature	The first index created for a table may not be dropped.
210021	85001	Progress/SQL stored procedure compilation	Location of the Java compiler was not specified.
210044	86008	OpenEdge stored procedure execution	Need to recompile stored procedures (run scriptSQLConvertSPTP - refer to release notes).
210045	86009	OpenEdge SQL triggers	Need to recompile triggers (run script SQLConvertSPTP - refer to release notes).
210047	22P00	OpenEdge SQL Update Statistics	Table %s.%s at Rowid %s has column %s whose value exceeding its max length or precision.
210048	70101	Data exception	Cache overflowed.
210049	22566	Data exception	Unable to read sequence record.
210050	22564	Data exception	The sequence was unable to cycle to another value.
210051	22563	Data exception	Sequence not found.
210052	22P00	Data exception	Maximum number of sequences already defined.

Error code	SQLSTATE value	Class condition	Subclass message
210054	2250z	Data exception	A sequence value was referenced outside of the defined range of values.
210055	42807	Access rule violation	Operation not allowed on the read-only database.
210056	42700	Syntax error	Syntax error at or about %s.
210057	85001	OpenEdge SQL Stored proc/Trigger	OpenEdge/SQL Java Native Interface(JNI) version not supported.
210058	85001	OpenEdge SQL Stored proc/Trigger	Error from Java compiler. Compiler messages follow.
210059	22P00	OpenEdge SQL statistics	Database statistics (%s) updated for all tables.
210060	22P00	OpenEdge SQL statistics	Database statistics (%s) updated for table %s.
210061	22P00	OpenEdge SQL statistics	Database statistics (%s) updated by direct user SQL statement (%s).
210062	70101	OpenEdge SQL statement mgr	mgr removed a prepared, never executed statement from statement cache. %s statements currently in use (%s cache).
210158	22555	Data exception	Incorrect duration supplied for given periodicity.
210159	22555	Data exception	Specify value for %s within limit %s
211013	3F001	Bad schema reference	SQL cannot alter or drop a table or index created by ABL or SQL 89.
211014	3F002	Bad schema reference	Incorrect view owner name on CREATE VIEW—cannot be PUB or _FOREIGN.
211015	3F003	Bad schema reference	Database object (table, view, index, trigger, procedure, or synonym) owned by "sysprogress" cannot be created, dropped, or altered.
211016	3F004	Bad schema reference	Database schema table cannot be created, dropped, or altered.
211017	3F004	Bad schema reference	Attempt to insert, update, or delete a row in a schema table.
211018	0A000	Array reference error	Array reference/update incorrect.

Error code	SQLSTATE value	Class condition	Subclass message
218001	P8P18	OpenEdge I18N NLS error	Failure to create a NLS character set conversion handler.
219901	P0000	Internal error	Internal error <*error_num1*> <*error_meaning*> in SQL from subsystem <*subsystem_name*> function <*function_name*> called from <*calling_function*> on <*object_2*> for <*object_1*>. Save log for Progress technical support.
219902	P0001	Internal error	Failure reading schema during DDL operation.
219903	P0002	Internal error	Inconsistent metadata - contact Progress technical support.
219907	22542	Data exception	Can not switch AUS logging state while AUS service is disabled.
219951	40P00	Transaction rollback	Fatal error <*error_num*> <*error_meaning*> in SQL from subsystem <*subsystem_name*> function <*function_name*> called from <*calling_function*> on <*object_2*> for <*object_1*>. Save log for Progress technical support.

5

OpenEdge SQL system limits

This topic provides a list of the maximum sizes for various attributes of the OpenEdge® SQL database environment, and for elements of SQL queries addressed to this environment.

The following table provides a list of the maximum sizes for various attributes of the OpenEdge SQL database environment, and for elements of SQL queries addressed to this environment.

Table 3: OpenEdge SQL system limits

Attribute	Value
Maximum number of open cursors	50
Maximum number of procedure arguments in an SQL CALL statement	100
Maximum length of an SQL statement	131000
Maximum length of a column in a table	31983
Maximum length of default value specification	250
Maximum length of a connect string	100
Maximum length for a table name	32
Maximum length for an area name	32
Maximum length for a username in a connect string	32
Maximum length of an error message	256

Attribute	Value
Maximum number of columns in a table	5000
Maximum number of bytes that can be inserted in a large key entry	1980
Maximum number of bytes that can be inserted in a small key entry	193
Maximum length of a CHECK constraint clause	240
Maximum number of nesting levels in an SQL statement	25
Maximum number of table references in an SQL statement: other platforms	250
Maximum size of input parameters for an SQL statement	512
Maximum number of outer references in an SQL statement	50
Maximum nesting level for view references	25
Maximum number of check constraints in a table	1000 total constraints per table
Maximum number of foreign constraints in a table	1000 total constraints per table
Maximum LOB length	1 GB
Maximum number of databases that can be connected	16 (1 primary and 15 auxiliary)

6

OpenEdge SQL system catalog tables

OpenEdge SQL maintains a set of system tables for storing information about tables, columns, indexes, constraints, and privileges. This topic describes those system catalog tables.

For details, see the following topics:

- Overview of system catalog tables
- SYSTABLES
- SYSCOLUMNS
- SYSINDEXES
- SYSCALCTABLE
- SYSNCHARSTAT
- SYSCOLAUTH
- SYSCOLSTAT
- SYSCOLUMNS_FULL
- SYSDATATYPES
- SYSDATESTAT
- SYSDBAUTH
- SYSFLOATSTAT
- SYSIDXSTAT

- SYSINTSTAT
- SYSNUMSTAT
- SYSPROCBIN
- SYSPROCCOLUMNS
- SYSPROCEDURES
- SYSPROCTEXT
- SYSPROCAUTH
- SYSREALSTAT
- SYSSEQAUTH
- SYSSEQUENCES
- SYSSYNONYMS
- SYSTABAUTH
- SYSTABLES_FULL
- SYSTBLSTAT
- SYSTIMESTAT
- SYSTINYINTSTAT
- SYSTRIGCOLS
- SYSTRIGGER
- SYSTSSTAT
- SQL_PROPERTIES
- SYSSTATUSE
- SYSTSTZSTAT
- SYSNVARCHARSTAT
- SYSVIEWS
- SYS_CHKCOL_USAGE
- SYS_CHK_CONSTRS
- SYS_KEYCOL_USAGE
- SYS_REF_CONSTRS
- SYS_TBL_CONSTRS

Overview of system catalog tables

OpenEdge SQL maintains a set of system tables for storing information about tables, columns, indexes, constraints, and privileges.

All users have read access to the system catalog tables. SQL Data Definition Language (DDL) statements and GRANT and REVOKE statements modify system catalog tables. The system tables are modified in response to these statements, as the database evolves and changes.

The owner of the system tables is sysprogress. If you connect to a OpenEdge SQL environment with a username other than sysprogress, you must use the owner qualifier when you reference a system table in a SQL query. Alternatively, you can issue a SET SCHEMA sysprogress statement to set the default username for unqualified table names to sysprogress.

Core tables store information on the tables, columns, and indexes that make up the database. The remaining tables contain detailed information on database objects and statistical information.

The following table lists the system catalog tables in the same order that they are presented in following sections.

Table 4: System tables and descriptions

System table	Summary description
SYSTABLES	Core system table; one row for each TABLE in the database
SYSCOLUMNS	Core system table; one row for each COLUMN of each table in the database
SYSINDEXES	Core system table. One row for each component of each INDEX in the database
SYSCALCTABLE	A single row with a single column set to the value 100
SYSNCHARSTAT	One row for each CHARACTER column in the database
SYSCOLAUTH	One row for each column for each user holding privileges on the column
SYSCOLSTAT	Provides statistical information on data distribution
SYSCOLUMNS_FULL	Superset of information in core system table SYSCOLUMNS
SYSDATATYPES	Information on supported data types
SYSDATESTAT	One set of rows for each DATE column in the database
SYSDBAUTH	One row for each user with database-wide privileges
SYSFLOATSTAT	One set of rows for each FLOAT column in the database
SYSIDXSTAT	Information on indexes in the database
SYSINTSTAT	One set of rows for each INTEGER column in the database
SYSNUMSTAT	One set of rows for each NUMERIC column in the database
SYSPROCBIN	One row for each compiled Java stored procedure or trigger in the database
SYSPROCCOLUMNS	One row for each column in the result set of a stored procedure
SYSPROCEDURES	One row for each stored procedure in the database

System table	Summary description
SYSPROCTEXT	One row for each Java source code for a stored procedure or trigger in the database
SYSREALSTAT	One set of rows for each REAL column in the database
SYSSEQAUTH	One row for each unique user/sequence combination, holding sequence privileges on a sequence of the database
SYSSEQUENCES	View of OpenEdge schema table_sequence
SYSSMINTSTAT	One set of rows for each SMALLINT column in the database
SYSSYNONYMS	One row for each SYNONYM in the database
SYSTABAUTH	One row for each unique user/table combination holding table privileges on a table in the database
SYSTABLES_FULL	Superset of information in core system table SYSTABLES
SYSTBLSTAT	Contains statistics for user tables in the database
SYSTIMESTAT	One set of rows for each TIME column in the database
SYSTINYINTSTAT	One set of rows for each TINYINT column in the database
SYSTRIGCOLS	One row for each column specified in each trigger in the database
SYSTRIGGER	One row for each trigger in the database
SYSTSSTAT	One set of rows for each TIMESTAMP column in the database
SYSTSTZSTAT	One set of rows for each TIMESTAMP WITH TIME ZONE column in the database
SYSNVARCHARSTAT	One set of rows for each VARCHAR column in the database
SYSVIEWS	One row for each VIEW in the database
SYS_CHKCOL_USAGE	One row for each CHECK CONSTRAINT defined on a column in the database
SYS_CHK_CONSTRS	One row for each CHECK CONSTRAINT defined on a user table in the database
SYS_KEYCOL_USAGE	One row for each column in the database defined with a PRIMARY KEY or FOREIGN KEY
SYS_REF_CONSTRS	One row for each table in the database defined with a REFERENTIAL INTEGRITY CONSTRAINT
SYS_TBL_CONSTRS	One row for each CONSTRAINT defined on a table in the database

SYSTABLES

Contains one row for each table in the database.

The following table provides details of the SYSTABLES table.

Table 5: SYSTABLES core system table

Column name	Column data type	Column size
creator	VARCHAR	32
has_ccnstrs	VARCHAR	1
has_fcnstrs	VARCHAR	1
has_pcnstrs	VARCHAR	1
has_ucnstrs	VARCHAR	1
id	INTEGER	4
owner	VARCHAR	32
rssid	INTEGER	4
segid	INTEGER	4
tbl	VARCHAR	32
tbl_status	VARCHAR	1
tbltype	VARCHAR	1

SYSCOLUMNS

Contains one row for each column of every table in the database.

The following table provides details of the SYSCOLUMNS table.

Table 6: SYSCOLUMNS core system table

Column name	Column data type	Column size
charset	VARCHAR	32
col	VARCHAR	32
collation	VARCHAR	32
coltype	VARCHAR	10

Column name	Column data type	Column size
dflt_value	VARCHAR	250
id	INTEGER	4
nullflag	VARCHAR	1
owner	VARCHAR	32
scale	INTEGER	4
tbl	VARCHAR	32
width	INTEGER	4

SYSINDEXES

Contains one row for each component of an index in the database. For an index with n components, there will be *n* rows in this table.

The following table provides details of the SYSINDEXES table.

Table 7: SYSINDEXES core system table

Column name	Column data type	Column size
abbreviate	BIT	1
active	BIT	1
creator	VARCHAR	32
colname	VARCHAR	32
desc	VARCHAR	144
id	INTEGER	4
idxcompress	VARCHAR	1
idxmethod	VARCHAR	2
idxname	VARCHAR	32
idxorder	CHARACTER	1
idxowner	VARCHAR	32
idxsegid	INTEGER	4
idxseq	INTEGER	4

Column name	Column data type	Column size
ixcol_user_misc	VARCHAR	20
rssid	INTEGER	4
tbl	VARCHAR	32
tblowner	VARCHAR	32

SYSCALCTABLE

Contains exactly one row with a single column with a value of 100.

The following table provides details of the SYSCALCTABLE table.

Table 8: SYSCALCTABLE system table

Column name	Column data type	Column size
fld	INTEGER	4

SYSNCHARSTAT

Contains a set of rows for each column in the database with data type CHAR. Used by the optimizer, each row contains a sample of values in the column.

The following table provides details of the SYSNCHARSTAT table.

Table 9: SYSNCHARSTAT system table

Column name	Column data type	Column size
colid	INTEGER	4
tblid	INTEGER	4
property	INTEGER	4
attribute	INTEGER	4
value	VARCHAR	2000

SYSCOLAUTH

Contains one row for the update privileges held by users on individual columns of tables in the database.

The following table provides details of the SYSCOLAUTH table.

Table 10: SYSCOLAUTH system table

Column name	Column data type	Column size
AuthMask	BIGINT	8
col	VARCHAR	32
grantee	VARCHAR	32
grantor	VARCHAR	32
grantor_domainId	INTEGER	4
grantee_domainId	INTEGER	4
ref	VARCHAR	1
sel	VARCHAR	1
tbl	VARCHAR	32
tblowner	VARCHAR	32
SnapShotVer	INT64	4
upd	VARCHAR	1

SYSCOLSTAT

Provides statistical information on data distribution for columns in tables.

The following table provides details of the SYSCOLSTAT table.

Table 11: SYSCOLSTAT system table

Column name	Column data type	Column size
colid	INTEGER	4
tblid	INTEGER	4
property	INTEGER	4
attribute	INTEGER	4
value	INTEGER	4
val_ts	TIMESTAMP	8

SYSCOLUMNS_FULL

A superset of information in the SYSCOLUMNS core system table.

The following table provides details of the SYSCOLUMNS_FULL table.

Table 12: SYSCOLUMNS_FULL system table

Column name	Column data type	Column size
array_extent	INTEGER	4
charset	VARCHAR	32
col	VARCHAR	32
collation	VARCHAR	32
coltype	VARCHAR	20
col_label	VARCHAR	60
col_label_sa	VARCHAR	12
col_subtype	INTEGER	4
description	VARCHAR	144
dflt_value	VARCHAR	250
dflt_value_sa	VARCHAR	12
display_order	INTEGER	4
field_rpos	INTEGER	4
format	VARCHAR	60
format_sa	VARCHAR	12
help	VARCHAR	126
help_sa	VARCHAR	12
id	VARCHAR	4
label	VARCHAR	60
label_sa	VARCHAR	12
nullflag	CHARACTER	2
owner	VARCHAR	32

Column name	Column data type	Column size
scale	INTEGER	4
tbl	VARCHAR	32
user_misc	VARCHAR	20
valexp	VARCHAR	144
valmsg	VARCHAR	144
valmsg_sa	VARCHAR	12
view_as	VARCHAR	100
width	INTEGER	4

SYSDATATYPES

Contains information on each data type supported by the database.

The following table provides details of the SYSDATATYPES table.

Table 13: SYSDATATYPES system table

Column name	Column data type	Column size
autoincr	SMALLINT	2
casesensitive	SMALLINT	2
createparams	VARCHAR	32
datatype	SMALLINT	2
dhtypename	VARCHAR	32
literalprefix	VARCHAR	1
literalsuffix	VARCHAR	1
localtypename	VARCHAR	1
nullable	SMALLINT	2
odbcmoney	SMALLINT	2
searchable	SMALLINT	2

Column name	Column data type	Column size
typeprecision	INTEGER	4
unsignedattr	SMALLINT	2

SYSDATESTAT

Contains a set of rows for each column of data type DATE. Used by the optimizer, each row contains a sample of values in the column.

The following table provides details of the SYSDATESTAT table.

Table 14: SYSDATESTAT system table

Column name	Column data type	Column size
colid	INTEGER	4
tblid	INTEGER	4
value	DATE	4
property	INTEGER	4
attribute	INTEGER	4

SYSDBAUTH

Contains the database wide privileges held by users.

The following table provides details of the SYSDBAUTH table.

Table 15: SYSDBAUTH system table

Column name	Column data type	Column size
AuthMask	BIGINT	8
AuthVer	INTEGER	4
dba_acc	VARCHAR	1
grantee	VARCHAR	32
grantee_domainId	INTEGER	4
res_acc	VARCHAR	1

SYSFLOATSTAT

Contains one row for each column of data type FLOAT. Used by the optimizer, each row contains a sampling of values in the column.

The following table provides details of the SYSFLOATSTAT table.

Table 16: SYSFLOATSTAT system table

Column name	Column data type	Column size
colid	INTEGER	4
tblid	INTEGER	4
value	FLOAT	4
property	INTEGER	4
attribute	INTEGER	4

SYSIDXSTAT

Contains statistics for indexes in the database.

The following table provides details of the SYSIDXSTAT table.

Table 17: SYSIDXSTAT system table

Column name	Column data type	Column size
idxid	INTEGER	4
property	INTEGER	4
attribute	INTEGER	4
value	INTEGER	4
val_ts	TIMESTAMP	8
tblid	INTEGER	4

SYSINTSTAT

Contains one row for each column of data type INTEGER. Used by the optimizer, each row contains a sampling of values in the column.

The following table provides details of the SYSINTSTAT table.

Table 18: SYSINTSTAT system table (Continued)

Column name	Column data type	Column size
colid	INTEGER	4
tblid	INTEGER	4
value	INTEGER	4
property	INTEGER	4
attribute	INTEGER	4

SYSNUMSTAT

Contains one row for each column of data type NUMERIC. Used by the optimizer, each row contains a sampling of values in the column.

The following table provides details of the SYSNUMSTAT table.

Table 19: SYSNUMSTAT system table

Column name	Column data type	Column size
colid	INTEGER	4
tblid	INTEGER	4
value	NUMERIC	32
property	INTEGER	4
attribute	INTEGER	4

SYSPROCBIN

Contains one or more rows for each stored procedure and trigger in the database. Each row contains compiled Java bytecode for its procedure or trigger.

The following table provides details of the SYSPROCBIN table.

Table 20: SYSPROCBIN system table

Column name	Column data type	Column size
id	INTEGER	4
proc_bin	VARBINARY	2000
proc_type	CHARACTER	2

Column name	Column data type	Column size
rssid	INTEGER	4
seq	INTEGER	4

SYSPROCCOLUMNS

Contains one row for each column in the result set of a stored procedure.

The following table provides details of the SYSPROCCOLUMNS table.

Table 21: SYSPROCCOLUMNS system table

Column name	Column data type	Column size
argtype	VARCHAR	32
col	VARCHAR	32
datatype	VARCHAR	32
dflt_value	VARCHAR	250
id	INTEGER	4
nullflag	CHAR	1
proc_id	INTEGER	4
rssid	INTEGER	4
scale	INTEGER	4
width	INTEGER	4

SYSPROCEDURES

Contains one row for each stored procedure in the database.

The following table provides details of the SYSPROCEDURES table.

Table 22: SYSPROCEDURES system table

Column name	Column data type	Column size
creator	VARCHAR	32
has_resultset	CHARACTER	1
has_return_val	CHARACTER	1

Column name	Column data type	Column size
owner	VARCHAR	32
proc_id	INTEGER	4
proc_name	VARCHAR	32
proc_type	VARCHAR	32
rssid	INTEGER	4
owner_domainid	INTEGER	8
creator_domainid	INTEGER	8

SYSPROCTEXT

Contains one or more rows for each stored procedure and trigger in the database. The row contains the Java source code for a procedure or trigger.

The following table provides details of the SYSPROCTEXT table.

Table 23: SYSPROCTEXT system table

Column name	Column data type	Column size
id	INTEGER	4
proc_text	VARCHAR	2000
proc_type	CHARACTER	2
rssid	INTEGER	4
seq	INTEGER	4

SYSPROCAUTH

Contains one or more rows for each stored procedure or User Defined Function which describes the user privileges to access these objects.

The following table provides details of the SYSPROCAUTH table.

Table 24: SYSPROCAUTH system table

Column name	Column data type	Column size
Alt	VARCHAR	1
AuthMask	BIGINT	8

Column name	Column data type	Column size
Exe	VARCHAR	1
grantor	VARCHAR	32
grantee	VARCHAR	32
grantor-domainId	INTEGER	8
grantee-domainId	INTEGER	8
procowner	VARCHAR	32
procname	VARCHAR	32
SnapShotVer	INTEGER	4

SYSREALSTAT

Contains one row for each column of data type REAL. Used by the optimizer, each row contains a sampling of values in the column.

The following table provides details of the SYSREALSTAT table.

Table 25: SYSREALSTAT system table

Column name	Column data type	Column size
colid	INTEGER	4
tblid	INTEGER	4
value	REAL	4
property	INTEGER	4
attribute	INTEGER	4

SYSSEQAUTH

Contains information about sequence privileges for database users.

The following table provides details of the SYSSEQAUTH table.

Table 26: SYSSEQAUTH system table

Column name	Column data type	Column size
Alt	VARCHAR	1

Column name	Column data type	Column size
AuthMask	BIGINT	8
grantee_domainId	INTEGER	4
grantor_domainId	INTEGER	4
grantee	VARCHAR	32
grantor	VARCHAR	32
ref	VARCHAR	1
sel	VARCHAR	1
seq-owner	VARCHAR	32
seq-name	VARCHAR	32
SnapShotVer	INTEGER	4
upd	VARCHAR	1
Dump	CHAR	1
Load	CHAR	1

SYSSEQUENCES

A view of the OpenEdge schema table_sequences.

The following table provides details of the SYSSEQUENCES table.

Table 27: SYSSEQUENCES system table

Column name	Column data type	Column size
seq-name	VARCHAR	32
seq-num	INTEGER	4
seq-init	INTEGER	4
seq-incr	INTEGER	4
seq-min	INTEGER	4
seq-max	INTEGER	4
cycle-ok	BIT	1

Column name	Column data type	Column size
seq-misc	VARCHAR	208
db-recod	INTEGER	4
user-misc	VARCHAR	20
seq-ownerdomainid	INTEGER	8
seq-owner	VARCHAR	32

SYSSYNONYMS

Contains one row for each synonym in the database.

The following table provides details of the SYSSYNONYMS table.

Table 28: SYSSYNONYMS system table

Column name	Column data type	Column size
ispublic	SMALLINT	2
screator	VARCHAR	32
sname	VARCHAR	32
sowner	VARCHAR	32
sremdb	VARCHAR	32
stbl	VARCHAR	32
sownerdomainid	INTEGER	8
screatordomainid	INTEGER	8
stblowner	VARCHAR	32

SYSTABAUTH

Contains information about table privileges for each user in the database.

The following table provides details of the SYSTABAUTH table.

Table 29: SYSTABAUTH system table

Column name	Column data type	Column size
alt	VARCHAR	1

Column name	Column data type	Column size
AuthMask	BIGINT	8
del	VARCHAR	1
exe	CHAR	1
grantee	VARCHAR	32
grantor	VARCHAR	32
grantor_domainId	INTEGER	4
grantee_domainId	INTEGER	4
ins	VARCHAR	1
ndx	VARCHAR	1
ref	VARCHAR	1
sel	VARCHAR	1
tbl	VARCHAR	32
tblowner	VARCHAR	32
upd	VARCHAR	1
Dump	CHAR	1
Load	CHAR	1
SnapShotVer	INTEGER	4

SYSTABLES_FULL

A superset of information in the SYSTABLES core system table.

The following table provides details of the SYSTABLES_FULL table.

Table 30: SYSTABLES_FULL system table

Column name	Column data type	Column size
can_dump	VARCHAR	126
can_load	VARCHAR	126
creator	VARCHAR	32

Column name	Column data type	Column size
description	VARCHAR	144
dump_name	VARCHAR	16
file_label	VARCHAR	60
file_label_sa	VARCHAR	12
frozen	BIT	1
has_ccnstrs	VARCHAR	1
has_fcnstrs	VARCHAR	1
has_pcnstrs	VARCHAR	1
has_ucnstrs	VARCHAR	1
hidden	BIT	1
id	INTEGER	4
last_change	INTEGER	4
owner	VARCHAR	32
prime_index	INTEGER	4
rssid	INTEGER	4
segid	INTEGER	4
tbl	VARCHAR	32
tbltype	VARCHAR	1
tbl_status	VARCHAR	1
user_misc	VARCHAR	20
valexp	VARCHAR	144
valmsg	VARCHAR	144
valmsg_sa	VARCHAR	12

SYSTBLSTAT

Contains statistics for tables.

The following table provides details of the SYSTBLSTAT table.

Table 31: SYSTBLSTAT system table

Column name	Column data type	Column size
property	INTEGER	4
attribute	INTEGER	4
value	INTEGER	4
val-ts	TIMESTAMP	8
tblid	INTEGER	4

SYSTIMESTAT

Contains one row for each column of data type TIME. Used by the optimizer, each row contains a sampling of values in the column.

The following table provides details of the SYSTIMESTAT table.

Table 32: SYSTIMESTAT system table

Column name	Column data type	Column size
colid	INTEGER	4
tblid	INTEGER	4
value	TIME	4
property	INTEGER	4
attribute	INTEGER	4

SYSTINYINTSTAT

Contains one row for each column of data type TINYINT. Used by the optimizer, each row contains a sampling of values in the column.

The following table provides details of the SYSTINYINTSTAT table.

Table 33: SYSTINYINTSTAT system table

Column name	Column data type	Column size
colid	INTEGER	4
tblid	INTEGER	4

Column name	Column data type	Column size
value	TINYINT	1
property	INTEGER	4
attribute	INTEGER	4

SYSTRIGCOLS

Contains one row for each column specified in each UPDATE trigger in the database.

The following table provides details of the SYSTRIGCOLS table.

Table 34: SYSTRIGCOLS system table

Column name	Column data type	Column size
colid	INTEGER	4
owner	VARCHAR	32
triggername	VARCHAR	32

SYSTRIGGER

Contains one row for each trigger in the database.

The following table provides details of the SYSTRIGGER table.

Table 35: SYSTRIGGER system table

Column name	Column data type	Column size
fire_4gl	BIT	1
owner	VARCHAR	32
refers_to_new	CHAR	1
refers_to_old	CHAR	1
rssid	INTEGER	4
statement_or_row	CHAR	1
tbl	VARCHAR	32
tblowner	VARCHAR	32
triggerid	INTEGER	4

Column name	Column data type	Column size
triggername	VARCHAR	32
trigger_event	VARCHAR	1
trigger_time	VARCHAR	1
owner_domain_id	INTEGER	8

SYSTSSTAT

Contains one row for each column of data type TIMESTAMP. Used by the optimizer, each row contains a sampling of values in the column.

The following table provides details of the SYSTSSTAT table.

Table 36: SYSTSSTAT system table

Column name	Column data type	Column size
colid	INTEGER	4
tblid	INTEGER	4
value	TIMESTAMP	8
property	INTEGER	4
attribute	INTEGER	4

SQL_PROPERTIES

The `SQL_PROPERTIES` system table contains the configuration details of the AUS Maintenance Window.

Here are the details of the `SQL_PROPERTIES` table.

_Property	_Attribute	_Value_n	_Value_ch	Default
AUS_Configuration_version	0	N	-	
AUS_Enable	0	0/1	-	0
AUS_LOG_Enable	0	1/0	-	0
AUS_Mode	0	1-Maintenance Window	-	1
AUS_Mode		2-Continues Mode	-	
AUS_MWindow	0		StartTime in char	1st Jan 1970 00:00:00
AUS_MWindow	1	Length of Maintenance Window in hrs		1hrs
AUS_MWindow	2	Cycle Length of Maintenance Window (time during two maintenance Window) in hrs.		24hrs
AUS_Throttle	0	n-Throttling level (0-#)		0
AUS_MarkPositive_percentage (Predicted and Experience cardinality difference)	0	>1		10%
AUS_Accessed_positive_percentage	0	>1		10%
AUS_Min_table_size	0	0-N		0
AUS_Max_table_size	0	-1-N (-1 infinity)		-1
AUS_Columstat_sample_percentage	0	N		100
AUS_TABLE_Level_config	0	TableID		
AUS_TABLE_Level_config	1	0 Disable 1 Table Stat 2 Index Stat 4 Column Stat		

SYSSTATUSE

Contains the table metadata or statistics for Autonomous Update Statistics (AUS).

Here are the details of the SYSSTATUSE table.

Column	Type	Default
Table Id	dh_132_t	
Partition Id	dh_132_t	
Column Id	dh_132_t	
Property	dh_132_t	
Attribute	dh_132_t	
History	dh_132_t	Null

Column	Type	Default
Value	BIGINT	Null
Value_ts	TIMESTAMP	Null
Misc	Varchar (50)	Null
Misc1	Varchar (50)	Null

SYSTSTZSTAT

Contains one row for each column of data type TIMESTAMP WITH TIME ZONE. Used by the optimizer, each row contains a sampling of values in the column.

The following table provides details of the SYSTSTZSTAT table.

Table 37: SYSTSTZSTAT system table

Column name	Column data type	Column size
colid	INTEGER	4
tblid	INTEGER	4
value	TIMESTAMP WITH TIME ZONE	12
property	INTEGER	4
attribute	INTEGER	4

SYSNVARCHARSTAT

Contains one row for each column of data type VARCHAR. Used by the optimizer, each row contains a sampling of values in the column.

The following table provides details of the SYSNVARCHARSTAT table.

Table 38: SYSNVARCHARSTAT system table

Column name	Column data type	Column size
colid	INTEGER	4
tblid	INTEGER	4
value	VARCHAR	2000
property	INTEGER	4
attribute	INTEGER	4

SYSVIEWS

Contains one row for each VIEW in the database.

The following table provides details of the SYSVIEWS table.

Table 39: SYSVIEWS system table

Column name	Column data type	Column size
creator	VARCHAR	32
owner	VARCHAR	32
seq	INTEGER	4
viewname	VARCHAR	32
viewtext	VARCHAR	2000

SYS_CHKCOL_USAGE

Contains one row for each column on which a check constraint is specified.

The following table provides details of the SYS_CHKCOL_USAGE table.

Table 40: SYS_CHKCOL_USAGE system table

Column name	Column data type	Column size
cnstrname	VARCHAR	32
colname	VARCHAR	32
owner	VARCHAR	32
tblname	VARCHAR	32

SYS_CHK_CONSTRS

Contains one row for each CHECK CONSTRAINT specified on a user table. The chkclause column contains the content of the CHECK clause.

The following table provides details of the SYS_CHK_CONSTRS table.

Table 41: SYS_CHK_CONSTRS system table

Column name	Column data type	Column size
chkclause	VARCHAR	2000

Column name	Column data type	Column size
chkseq	INTEGER	4
cnstrname	VARCHAR	32
owner	VARCHAR	32
tblname	VARCHAR	32

SYS_KEYCOL_USAGE

Contains one row for each column on which a PRIMARY KEY or FOREIGN KEY is specified.

The following table provides details of the SYS_KEYCOL_USAGE table.

Table 42: SYS_KEYCOL_USAGE system table

Column name	Column data type	Column size
cnstrname	VARCHAR	32
colname	VARCHAR	32
colposition	INTEGER	4
owner	VARCHAR	32
tblname	VARCHAR	32

SYS_REF_CONSTRS

Contains one row for each REFERENTIAL INTEGRITY CONSTRAINT specified on a user table.

The following table provides details of the SYS_REF_CONSTRS table.

Table 43: SYS_REF_CONSTRS system table

Column name	Column data type	Column size
cnstrname	VARCHAR	32
deleterule	VARCHAR	1
owner	VARCHAR	32
refcnstrname	VARCHAR	32
refowner	VARCHAR	32

Column name	Column data type	Column size
reftblname	VARCHAR	32
tblname	VARCHAR	32

SYS_TBL_CONSTRS

Contains one row for each table constraint in the database.

The following table provides details of the SYS_TBL_CONSTRS table.

Table 44: SYS_TBL_CONSTRS system table

Column name	Column data type	Column size
cnstrname	VARCHAR	32
cnstrtype	VARCHAR	1
idxname	VARCHAR	32
owner	VARCHAR	32
tblname	VARCHAR	32

7

OpenEdge SQL language elements

This topic describes Standard SQL language elements that are common to OpenEdge® SQL.

For details, see the following topics:

- OpenEdge SQL identifiers
- Number formats
- Date-time formats
- Date formats
- Time formats
- Data types

OpenEdge SQL identifiers

Identifiers are user-specified names for elements such as tables, views, and columns. The maximum length for SQL identifiers is 32 characters.

The two types of SQL identifiers are:

- Conventional identifiers
- Delimited identifiers enclosed in double quotation marks

Conventional identifiers

Conventional SQL identifiers must:

- Begin with an uppercase or lowercase letter
- Contain only letters (A-Z), digits (0-9), or the underscore character (_)
- Not be reserved words, such as CREATE or DROP
- Use ASCII characters only

SQL does not distinguish between uppercase and lowercase letters in SQL identifiers. It converts all names specified as conventional identifiers to uppercase, but statements can refer to the names in mixed case.

Example

The following example illustrates the use of identifiers in a simple query statement where CustNum, Order, and OrderDate are the user-specified names of columns:

```
SELECT CustNum, COUNT(*)
     FROM Order
     WHERE OrderDate < TO_DATE ('3/31/2004')
     GROUP BY CustNum
     HAVING COUNT (*) > 10 ;
```

Delimited identifiers

Delimited identifiers are strings of no more than 32 ASCII characters enclosed in quotation marks (" "). Delimited identifiers allow you to create identifiers that are identical to keywords or that use special characters (such as #, &, or *) or a space.

Enclosing a name in quotation marks preserves the case of the name and allows it to be a reserved word or to contain special characters. Special characters are any characters other than letters, digits, or the underscore character. Subsequent references to a delimited identifier must also use quotation marks. To include a quotation mark character in a delimited identifier, precede it with another quotation mark.

The following code example uses a delimited identifier to create a table named "Dealer Table", where the space character is part of the name:

```
CREATE TABLE "Dealer Table" (name, address, city, state)
     AS
          SELECT name, address, city, state
          FROM customer
          WHERE state IN ('CA','NY', 'TX') ;
```

Number formats

Numeric data has cultural characteristics that international applications must address. For example, numeric separators (decimal and thousands separators) and currency symbols differ across locales and regions. Therefore, OpenEdge applications provide the capability to store, manage and display data in formats that meet the needs of the international market. The following table defines the number formats that are supported by OpenEdge SQL.

Table 45: OpenEdge SQL Number Formats

Format	Example	Definition
$	$999	Returns a value with a leading dollar sign.
,	9,999	Returns a comma in the specified position (not a thousands separator).
.	99.99	Returns a decimal in the specified position (not a fractional indicator).
0	0999 9990	Displays and positions a leading or trailing zero.
9	9999	Sets the number of significant digits to be displayed. Displays the leading space if positive, leading minus if negative. Leading zeros are blank except for a zero value returning a zero for the integer part of the number.
D	99D9	Returns NLS_NUMERIC_CHARACTER in the specified position. The default D character is (.).
G	9G99	Returns NLS_NUMERIC_CHARACTER in the specified position. The default G character is (,).
L	L999	Return the local currency symbol NLS_CURRENCY in the specified position.

Date-time formats

The TO_CHAR function supports the date-format and the time-format strings to control the output of date and time values. The format strings consist of keywords that SQL interprets and replaces with formatted values.

Syntax

```
TO_CHAR ( expresion [ , format_string ] )
```

Parameters

expression

Converts to character form. It must evaluate to a value of the date or time data type to use the format_string.

format_string

Specifies the format of the output. SQL ignores the format string if the expression argument does not evaluate to a date or time.

Supply the format strings, enclosed in single quotation marks, as the second argument to the function. The format strings are case sensitive. For instance, SQL replaces DAY with all uppercase letters, but follows the case of Day.

Example

The following example illustrates the difference between how a date value displays with and without the TO_CHAR function:

```
SELECT C1 FROM T2;
C1
--
09/29/1952
1 record selected
  SELECT TO_CHAR(C1, 'Day, Month ddth'),
     TO_CHAR(C2, 'HH12 a.m.') FROM T2;
 TO_CHAR(C1,DAY, MONTH DDTH) TO_CHAR(C2,HH12 A.M.)
--------------------------- --------------------
Monday  , September 29th  02 p.m.
1 record selected
```

Date formats

A date-format string can contain any of the following format keywords along with other characters. The format keywords in the format string are replaced by corresponding values to get the result. The other characters are displayed as literals.

The following table lists the date formats and their corresponding descriptions.

Table 46: Date formats and descriptions

Date format	Description
CC	The century as a two-digit number.
YYYY	The year as a four-digit number.
YYY	The last three digits of the year.
YY	The last two digits of the year.
Y	The last digit of the year.
Y,YYY	The year as a four-digit number with a comma after the first digit.
Q	The quarter of the year as a one-digit number (with values 1, 2, 3, or 4).
MM	The month value as a two-digit number (in the range 01-12).
MONTH	The name of the month as a string of nine characters ('JANUARY' to 'DECEMBER').
MON	The first three characters of the name of the month (in the range'JAN' to 'DEC').
WW	The week of the year as a two-digit number (in the range 01-53).

Date format	Description
W	The week of the month as a one-digit number (in the range 1-5).
DDD	The day of the year as a three-digit number (in the range 001-366).
DD	The day of the month as a two-digit number (in the range 01-31).
D	The day of the week as a one-digit number (in the range 1-7, 1 for Sunday and 7 for Saturday).
DAY	The day of the week as a character string of nine characters (in the range 'SUNDAY' to 'SATURDAY').
DY	The day of the week as a character string of three characters (in the range 'SUN' to 'SAT').
J	The Julian day (number of days since DEC 31, 1899) as an eight-digit number.
TH	When added to a format keyword that results in a number, this format keyword ('TH') is replaced by the string 'ST', 'ND', 'RD', or 'TH' depending on the last digit of the number.

Example

The following example illustrates the use of the DAY, MONTH, DD, and TH format strings:

```
SELECT C1 FROM T2;
 C1
--
09/29/1952
 1 record selected
  SELECT TO_CHAR (C1, 'Day, Month ddth'),
    TO_CHAR (C2, 'HH12 a.m.') FROM T2;
 TO_CHAR (C1,DAY, MONTH DDTH) TO_CHAR (C2,HH12 A.M.)
---------------------------- ---------------------
Monday, September 29th 02 p.m.
 1 record selected
```

Time formats

A time format string can contain any of the following format keywords along with other characters. The format keywords in the format string are replaced by corresponding values to get the result. The other characters are displayed as literals.

The following table lists the time formats and their corresponding descriptions.

Table 47: Time formats and descriptions

Time format	Description
AM PM	The string AM or PM depending on whether time corresponds to morning or afternoon
A.M. P.M.	The string A.M. or P.M. depending on whether time corresponds to morning or afternoon
HH12	The hour value as a two-digit number (in the range 00 to 11)

Time format	Description
HH HH24	The hour value as a two-digit number (in the range 00 to 23)
MI	The minute value as a two-digit number (in the range 00 to 59)
SS	The seconds value as a two-digit number (in the range 00 to 59)
SSSSS	The seconds from midnight as a five-digit number (in the range 00000 to 86399)
MLS	The milliseconds value as a three-digit number (in the range 000 to 999)

Example

The following example illustrates the TO_CHAR function, and the Day, Month, dd, and HH12 format strings:

```
SELECT C1 FROM T2;
 C1
--
09/29/1952
 1 record selected
  SELECT TO_CHAR (C1, 'Day, Month ddth'),
   TO_CHAR (C2, 'HH12 a.m.') FROM T2;
 TO_CHAR (C1,DAY, MONTH DDTH) TO_CHAR (C2,HH12 A.M.)
---------------------------- ---------------------
Monday , September 29th 02 p.m.
1 record selected
```

Data types

CREATE TABLE statements specify the data type for each column in the table they define. This section describes the data types SQL supports for table columns. All the data types can store null values. A null value indicates that the value is not known and is distinct from all non-null values.

Syntax

```
char_data_type
   | exact_numeric_data_type
   | approx_numeric_data_type
   | date_time_data_type
   | bit_string_data_type
   | array_data_type
   | vararray_data_type
```

Example

The following example illustrates the use of data types in a CREATE TABLE statement:

```
CREATE TABLE CUSTOMERS
   (CUST_NUM INTEGER NOT NULL,
     COMPANY VARCHAR (20) NOT NULL,
    CUST_REP INTEGER,
CREDIT_LIMIT INTEGER,
 PRIMARY KEY (CUST_NUM));
```

The OpenEdge SQL data types are:

- CHARACTER
- EXACT NUMERIC
- APPROXIMATE NUMERIC
- DATE-TIME
- BIT STRING
- ARRAY

Each data type is described in the following sections.

Character data types

Character data strings consist of a sequence of character from a defined character set, such as ASCII. A character string may have a fixed or varying length.

Syntax

This is the syntax for character data types:

```
{ CHARACTER | CHAR }[ ( length ) ]|
{ CHARACTER VARYING | CHAR VARYING | VARCHAR | LVARCHAR | CLOB }[ ( length ) ]
```

Parameters

CHARACTER [(*length*)]

CHARACTER (alias CHAR) corresponds to a null-terminated character string with the length specified. Values are padded with blanks to the specified length. The default length is 1. The maximum length is 2,000 characters.

The OpenEdge SQL representation is a variable-length string. The host language representation is equivalent to a C language character string.

{ CHARACTER VARYING | CHAR VARYING | VARCHAR | CLOB | LVARCHAR }[(*length*)]

CHARACTER VARYING, CHAR VARYING, and VARCHAR corresponds to a variable-length character string with the maximum length specified. The default length is 1 character. The maximum length is 31,995 characters. CLOBhas a maximum length of 1,073,741,823. A CLOB is an object of data type CLOB.

Notes

- For data types CHARACTER(length) and VARCHAR(length) the value of length specifies the number of characters.
- The maximum length can be as large as 31,995. The sum of all the column lengths of a table row must not exceed 31,960.
- Due to index size limitations, only the narrower VARCHAR columns can be indexed.

Maximum length for VARCHAR

The maximum length of the VARCHAR data type depends on:

- **The number of columns in a table** — More columns in a table further limits the length of VARCHAR data.
- **When a table was created** — Tables created earlier can support longer VARCHAR data than tables created later.

National Language Support (NLS)

The VARCHAR data type has NLS. The choice of character set affects the available character count or maximum length of the data column. The limits established above assume a single-byte character set. Using a multiple-byte character set lowers the maximum character count proportionally. For example, if all the characters in a character set take 3 bytes per character, the practical maximum is 10,660 (31,982 divided by 3). If, however, you are using a variable-width character set, you will be able to hold between 10,660 and 31,982 characters, depending on the actual mix of characters you use.

Concatenation operator

Use the concatenation operator (||) to join two text strings together.

The following example provides an example of a concatenation operator used in a query:

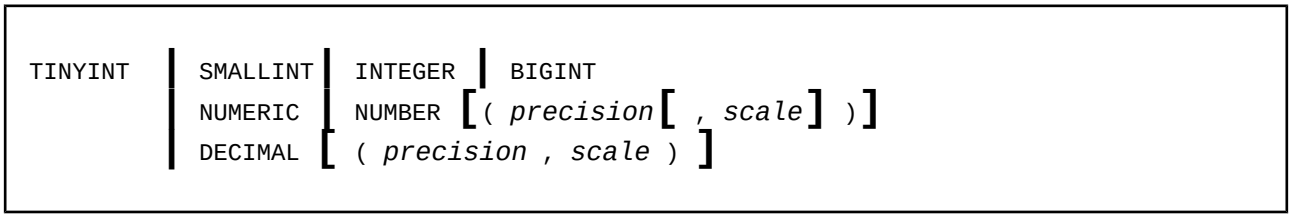
Example

```
SELECT firstname  || ' ' || lastname  from Employee;
```

Exact numeric data types

Exact numeric data types are used to represent the exact value of a number. This is the syntax for exact numeric data types:

Syntax



Parameters

`TINYINT`

Corresponds to an integer value in the range -128 to +127 inclusive.

`SMALLINT`

Corresponds to an integer value in the range of -32768 to 32767 inclusive.

`INTEGER`

Corresponds to an integer value in the range of -2147483648 to 2147483647 inclusive.

`BIGINT`

Corresponds to an integer value in the range of -9223372036854775808 to 9223372036854775807 inclusive.

`NUMERIC | NUMBER [ ( precision [ , scale ] ) ]`

Corresponds to a number with the given precision (maximum number of digits) and scale (the number of digits to the right of the decimal point). By default, `NUMERIC` columns have a precision of 32 and a scale of 0. If `NUMERIC` columns omit the scale, the default scale is 0.

The range of values for a `NUMERIC` type column is `-n` to `+n` where `n` is the largest number that can be represented with the specified precision and scale. If a value exceeds the precision of a `NUMERIC` column, SQL generates an overflow error. If a value exceeds the scale of a `NUMERIC` column, SQL rounds the value.

`NUMERIC` type columns cannot specify a negative scale or specify a scale larger than the precision.

`DECIMAL [ ( precision , scale ) ]`

Equivalent to type `NUMERIC`.

Approximate numeric data types

Approximate numeric data types are used to define data with a wide range of values and whose precision does not have to be exact.

Syntax

This is the syntax for an approximate data type:

```
{ REAL | DOUBLE PRECISION | FLOAT [ ( precision ) ]}
```

Parameters

REAL

Corresponds to a single precision floating-point number equivalent to the C language float type.

DOUBLE PRECISION

Corresponds to a double precision floating-point number equivalent to the C language double type.

FLOAT [(*precision*)]

Corresponds to a double precision floating-point number of the given precision, in bytes. By default, FLOAT columns have a precision of 8. The REAL data type is same as a FLOAT(4), and double-precision is the same as a FLOAT(8).

Date-time data types

Date-time data types are used to define points in time.

Syntax

This is the syntax for the date-time data types:

```
DATE | TIME | TIMESTAMP | TIMESTAMP WITH TIME ZONE
```

Parameters

DATE

Stores a date value as three parts: year, month, and day. The ranges for the parts are:

- Year: 1 to 9999
- Month: 1 to 12
- Day: Lower limit is 1; the upper limit depends on the month and the year

TIME

Stores a time value as four parts: hours, minutes, seconds, and milliseconds. The ranges for the parts are:

- Hours: 0 to 23
- Minutes: 0 to 59
- Seconds: 0 to 59

- Milliseconds: 0 to 999

`TIMESTAMP`

Combines the parts of `DATE` and `TIME`

`TIMESTAMP WITH TIME ZONE`

Combines the elements of `TIMESTAMP` with a time zone offset

Bit string data types

Bit string data types are used to define bit strings, which are sequences of bits having the value of either 0 or 1.

Syntax

This is the syntax for a bit string data type:

```
BIT | BINARY | VARBINARY | BLOB [ ( length ) ]
```

Parameters

`BIT`

Corresponds to a single bit value of 0 or 1.

SQL statements can assign and compare values in `BIT` columns to and from columns of types `CHAR`, `VARCHAR`, `BINARY`, `VARBINARY`, `TINYINT`, `SMALLINT`, and `INTEGER`. However, in assignments from `BINARY` and `VARBINARY`, the value of the first four bits must be `0001` or `0000`.

No arithmetic operations are allowed on `BIT` columns.

`BINARY [ ( length ) ]`

Corresponds to a bit field of the specified length of bytes. The default length is 1 byte. The maximum length is 2000 bytes.

When inserting literals into binary data types, `INSERT` statements must use a special format to store values in `BINARY` columns. They can specify the binary values as a bit string, hexadecimal string, or character string. `INSERT` statements must enclose binary values in single-quote marks, preceded by b for a bit string and x for a hexadecimal string.

The following table lists the specification formats for binary values.

Table 48: Specification formats for binary values

Specification	Format	Example
Bit string	`b "`	`b '1010110100010000'`
Hexadecimal string	`x "`	`x 'ad10'`
Character string	"	'ad10'

SQL interprets a character string as the character representation of a hexadecimal string.

If the data inserted into a BINARY column is less than the length specified, SQL pads it with zeros.

BINARY data can be assigned and compared to and from columns of type BIT, CHAR, and VARBINARY. Arithmetic operations are not allowed.

VARBINARY (*length*)

Corresponds to a variable-length bit field of the specified length in bytes. The default length is 1 byte. The maximum length is 31,995 bytes. The default length is 1. Due to index limitations, only the narrower VARBINARY columns can be indexed.

LVARBINARY/BLOB (*length*)

Corresponds to an arbitrarily long byte array with the maximum length defined by the amount of available disk storage up to 1,073,741,823. A BLOB is an object of data type LVARBINARY.

Maximum length for VARBINARY

The maximum length of the VARBINARY data type depends on:

- **The number of columns in a table** — More columns in a table further limits the length of VARBINARY data.
- **When a table was created** — Tables created earlier can support longer VARBINARY data than tables created later.

BLOB limitations

Current limitations for BLOB support are as follows:

- BLOB data columns cannot be part of an index.
- BLOB data columns cannot be used for variables or as parameters in stored procedures.
- Comparison operations are not supported on BLOB columns. Comparison operations between BLOB columns are not supported. Comparison operations between BLOB columns and columns of other data types are not supported.
- Conversion, aggregate, and scalar functions are disallowed on this data type.
- BLOB does not have National Language Support (NLS).
- The default size for a BLOB is 32000 bytes.

Language support for BLOB

This data type has normal column functionality except for the following exceptions:

- A column of data type BLOB is not a valid column name in a CREATE INDEX statement.
- When issuing a CREATE TABLE statement, a valid data type for the column definitions is BLOB. However, BLOB does not allow the column constraints of PRIMARY KEY, FOREIGN KEY, UNIQUE, REFERENCES, and CHECK.
- The VALUES option on the INSERT statement is not valid for the BLOB data type.
- In a SELECT statement, a WHERE, GROUP BY, HAVING, or ORDER BY clause cannot use a column of data type BLOB.

Utility support for BLOB

Use BINARY DUMP/LOAD to dump and load data that contains the BLOB data type. SQLDUMP and SQLLOAD do **not** support tables with BLOB column data.

CLOB limitations

Current limitations for BLOB support are as follows:

- CLOB data columns cannot be part of an index.
- CLOB data columns cannot be used for variables or as parameters in stored procedures.
- Comparison operations are not supported on CLOB columns. Comparison operations between CLOB columns are not supported. Comparison operations between CLOB columns and columns of other data types are not supported.
- The LIKE predicate is supported for CLOB columns.
- Conversion, aggregate, and scalar functions are disallowed on this data type.
- The default size for a CLOB is 32000 bytes.

Utility support for LVARCHAR (CLOB)

Use BINARY DUMP/LOAD to dump and load data that contains the CLOB data type. SQLDUMP and SQLLOAD do **not** support tables with CLOBS column data.

Array data types

The ARRAY data type is a composite data value that consists of zero or more elements of a specified data type (known as the element type). VARARRAY data type allows the size of an individual element value to exceed its declared size as long as the total size of the array is smaller than the array's SQL width.

The VARARRAY type is most compatible with the ABL array data definitions. For best compatibility with the ABL, use the VARARRAY type. The ARRAY type is less ABL compatible and more SQL standard compliant.

Syntax

This is the syntax for the array data type:

```
data_type ARRAY[int] | VARARRAY[int]
```

Parameters

data_type

The data type of the array. This is also known as the element type.

Supported data types are: BINARY, BIT, CHAR, VARCHAR, DATE, DECIMAL, DOUBLE PRECISION, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TIME, TIMESTAMP, TIMESTAMP_TZ, TINYINT, and VARBINARY.

[*int*]

An unsigned integer, indicating the array's maximum element size.

Notes

- OpenEdge SQL limits an array's size. The array's size must be an integer between 1 and 9999.
- Array columns and element references cannot be indexed because:
  - You cannot define a UNIQUE key with columns of type ARRAY.
  - You cannot define a PRIMARY key with columns of type ARRAY.
  - You cannot define a FOREIGN key with columns of type ARRAY.
- Array columns and element references cannot be used in GROUP BY clauses.

Example

In this example, table TBL is created. TBL has two columns: column C1 is an array of up to 3 elements, all of them type int and column C2 is a variable-sized array of up to 4 elements, all of them type varchar:

```
CREATE TABLE TBL (C1 int ARRAY[3], C2 varchar(5) VARARRAY[4]);
```

The size of any element in C2 can be up to 20 characters (5*4) with a total size of 20 characters.

ARRAY element reference

An element reference allows you to access a specific element of an array. It operates on two arguments: the first must evaluate to an array and the second must evaluate to an integer. The integer refers to the ordinal position of the element in the array (the first element in the array is element number one, the second is element number two, and so on).

It is possible to select the array as a whole value, rather than selecting individual array elements. When the array as a whole is selected, SQL returns a VARCHAR datatype value. That value comprises all the elements, converted to character form, with elements separated from each other by a ";" delimiter.

Example

In this example, the fourth element of the array column named array_column is returned:

```
SELECT array_column[4] FROM TBL;
```

Default value for ARRAY columns

When creating array columns, you can specify a default value.

Example

Since no value is specified for array column C2 when inserting values, the default value is used. The result returned from this example would be 10;10;10:

```
CREATE TABLE tbl (C1 int, C2 int ARRAY[3] default '10');
INSERT INTO tbl (C1) VALUES (1);
SELECT C2 FROM tble WHERE C1 = 1;
```

Note: The default value is applicable only at the column level. This means that if fewer values are specified when executing an insert statement, the default will not be used to fill up the rest of the array elements. Instead, NULL is used.

Assignment

When an array is assigned to an array target, the assignment is done one element at a time. Two arrays are assignable if their element's data types are mutually assignable. This means:

- When an array is taken from SQL data to be assigned to an array target, the number of elements in the source array equals the maximum number of elements in the target array. The value of each element of the source is assigned to the corresponding element of the target.
- If the maximum number of elements in the target array is less than the number of elements in the source array, then an error is returned.
- If the maximum number of elements in the target array is greater than the number of elements in the source array, the assignment of each of the source element values to the target elements occurs and the rest of the target elements will be assigned values of NULL.

Example

```
CREATE TABLE TBL (C1 int, C2 int ARRAY[3]);
INSERT into TBL values (1, '111;222;333');
UPDATE TBL SET C2 = '777;888;999';
```

Comparison

OpenEdge SQL provides two scalar comparison operators: = and <>. Two arrays are comparable if their element data types are mutually comparable. During comparison, the elements are compared pair-wise in element order. Two arrays are equal if:

- They both have the same number of elements
- Each pair of elements is equal

Two arrays are not equal if:

- They do not have the same number or elements
- At least one pair of elements is not equal

Literals

A literal, also called a constant, is a type of expression that specifies a constant value. Generally, you can specify a literal wherever SQL syntax allows an expression. Some SQL constructs allow literals but disallow other forms of expressions.

There are three types of literals:

- `NUMERIC`
- `CHARACTER-STRING`
- `DATE-TIME`

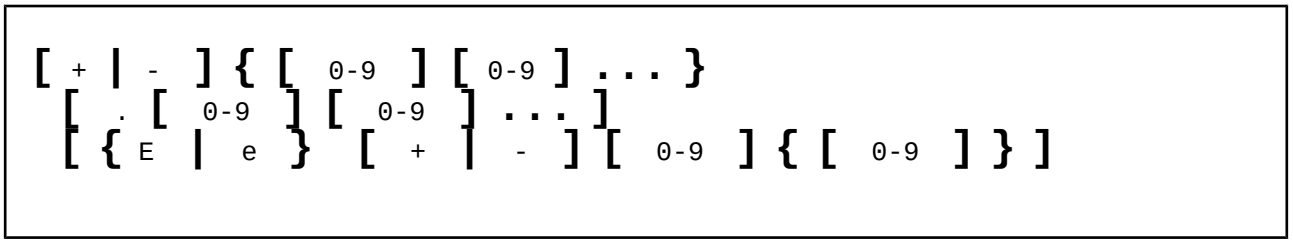
The following sections discuss each type of literal.

Numeric literals

A numeric literal is a string of digits that SQL interprets as a decimal number. SQL allows the string to be in a variety of formats, including scientific notation.

Syntax

This is the syntax for numeric literals:



Example

The numeric strings in the following example are all valid:

```
123
123.456
-123.456
12.34E-04
```

Character-string literals

A character-string literal is a string of characters enclosed in single quotation marks ('). To include a single quotation mark in a character-string literal, precede it with an additional single quotation mark.

The INSERT statements in the following example show embedding quotation marks in character-string literals:

```
insert into quote values('unquoted literal');
insert into quote values('''single-quoted literal''');
insert into quote values('"double-quoted literal"');
insert into quote values('O''Hare');select * from quote;
 c1
--
unquoted literal
'single-quoted literal'
"double-quoted literal"
O'Hare
 4 records selected
```

A character string literal can contain multi-byte characters in the character set used by the SQL client. Only single-byte ASCII-encoded quote marks are valid in the syntax.

Date-time literals

SQL supports special formats for literals to be used in conjunction with date-time data types. Basic predicates and the VALUES clause of INSERT statements can specify date literals directly for comparison and insertion into tables. In other cases, you need to convert date literals to the appropriate date-time data type with the CAST, CONVERT, or TO_DATE scalar functions.

Enclose date-time literals in single quotation marks ('').

Notes

- All text (names of days, months, ordinal number endings) in all date-format literals must be in the English language. The default date format is American. You can explicitly request another date format by using a format string.
- Time literals are in the English language only.

Date literals

A date literal specifies a day, month, and year using any of the following formats, enclosed in single quotation marks ('').

Syntax

This is the syntax for date literals:

```
{ d'yyyy-mm-dd'}
|mm-dd-yyyy| mm/dd/yyyy| mm-dd-yy| mm/dd/yy| yyyy-mm-dd
  | yyyy/mm/dd| dd-mon-yyyy| dd/mon/yyyy| dd-mon-yy| dd/mon/yy
```

Parameter

`{ d 'yyyy-mm-dd'}`

A date literal enclosed in an escape clause is compatible with ODBC. Precede the literal string with an open brace ({) and a lowercase d. End the literal with a close brace)}). If you use the ODBC escape clause, you must specify the date using the format *yyyy-mm-dd*.

Notes

Date literals must be enclosed in single quotations, such as the case with column values in an INSERT statement.

Example

The following example illustrates how to use the date literal format with an INSERT statement:

```
INSERT INTO dtest VALUES ( { d '2004-05-07' } )
```

The INSERT and SELECT statements in the following example show some of the supported formats for date literals:

```
CREATE TABLE T2 (C1 DATE, C2 TIME);
INSERT INTO T2 (C1) VALUES('5/7/56');
INSERT INTO T2 (C1) VALUES('7/MAY/1956');
INSERT INTO T2 (C1) VALUES('1956/05/07');
INSERT INTO T2 (C1) VALUES({d '1956-05-07'});
INSERT INTO T2 (C1) VALUES('29-SEP-1952');
 SELECT C1 FROM T2;
 --------------
c1
1956-05-07
1956-05-07
1956-05-07
1956-05-07
1952-09-29
```

Time literals

Time literals specify an hour, minute, second, and millisecond, using the following format, enclosed in single quotation marks (' ').

Syntax

This is the syntax for time literals:

```
{ t 'hh:mi:ss[:mls] }
```

Parameters

`{ t 'hh:mi:ss' }`

A time literal enclosed in an escape clause is compatible with ODBC. Precede the literal string with an open brace ({) and a lowercase t. End the literal with a close brace (}).

Note: If you use the ODBC escape clause, you must specify the time using the format *hh*:*mi*:*ss*.

hh

Specifies the hour value as a two-digit number in the range 00 to 23.

mi

Specifies the minute value as a two-digit number in the range `00` to `59`.

ss

Specifies the seconds value as a two-digit number in the range `00` to `59`.

mls

Specifies the milliseconds value as a three-digit number in the range `000` to `999`.

Examples

The following example illustrates how to use the time literal format with an `INSERT` statement:

```
INSERT INTO ttest VALUES ( { t '23:22:12' } ) ;
```

The `INSERT` statements in the following example show some of the formats SQL will and will not accept for time literals:

```
INSERT INTO T2 (C2) VALUES('3');
error(-20234): Invalid time string
 INSERT INTO T2 (C2) VALUES('8:30');
error(-20234): Invalid time string
 INSERT INTO T2 (C2) VALUES('8:30:1');
INSERT INTO T2 (C2) VALUES('8:30:');
error(-20234): Invalid time string
  INSERT INTO T2 (C2) VALUES('8:30:00');
INSERT INTO T2 (C2) VALUES('8:30:1:1');
INSERT INTO T2 (C2) VALUES({t '8:30:1:1'});
```

The `SELECT` statement in the following example illustrates which `INSERT` statements successfully inserted a row:

```
SELECT C2 FROM T2;c2
 c2
--
08:30:01
08:30:00
08:30:01
08:30:01
```

Timestamp literals

Timestamp literals specify a date and a time separated by a space, enclosed in single quotation marks (`''`).

Syntax

This is the syntax for timestamp literals:

```
{ ts 'yyyy-mm-dd hh:mi:ss' }
```

Parameters

`{ ts 'yyyy-mm-dd hh:mi:ss' }`

A timestamp literal enclosed in an escape clause is compatible with ODBC. Precede the literal string with an open brace ({) and a lowercase `ts`. End the literal with a close brace (}). Note that braces are part of the syntax. If you use the ODBC escape clause, you **must** specify the timestamp using the format *yyyy-mm-dd hh*:*mi*:*ss*.

date_literal

A date.

time_literal

A time literal.

Example

The following example illustrates how to `INSERT` a timestamp literal into a column:

```
INSERT INTO DTEST
VALUES ( { ts '1956-05-07 10:41:37'} ) ;
```

The following example illustrates a timestamp literal with the ODBC escape clause:

```
SELECT * FROM DTEST WHERE C1 = {ts '1985-08-10 05:41:37'} ;
```

Relational operators

Relational operators specify how SQL compares expressions in basic and quantified predicates.

Syntax

This is the syntax for relational operators:

```
= | <> | != | ^= | < | <= | > | >=
```

The following table lists the relational operators and the resulting predicates for each operator.

Table 49: Relational operators and resulting predicates

Relational operator	Predicate for this relational operator
=	True if the two expressions are equal.
<> | != | ^=	True if the two expressions are not equal. The operators != and ^= are equivalent to <>.
<	True if the first expression is less than the second expression.

Relational operator	Predicate for this relational operator
<=	True if the first expression is less than or equal to the second expression.
>	True if the first expression is greater than the second expression.
>=	True if the first expression is greater than or equal to the second expression.

Basic Predicate

A basic predicate compares two values using a relational operator. If a basic predicate specifies a query expression, then the query expression must return a single value. Basic predicates often specify an inner join.

If the value of any expression is null or the *query_expression* does not return any value, then the result of the predicate is set to false.

Syntax

This is the syntax for a basic predicate:

```
expression relop { expression | ( query_expression ) }
```

Quantified Predicate

The quantified predicate compares a value with a collection of values using a relational operator. A quantified predicate has the same form as a basic predicate with the *query_expression* being preceded by the ALL, ANY, or SOME keyword. The result table returned by *query_expression* can contain only a single column.

When you specify ALL, the predicate evaluates to true if the *query_expression* returns no values or the specified relationship is true for all the values returned.

When you specify SOME or ANY, the predicate evaluates to true if the specified relationship is true for at least one value returned by the *query_expression*. There is no difference between the SOME and ANY keywords. The predicate evaluates to false if the *query_expression* returns no values or if the specified relationship is false for all the values returned.

Syntax

This is the syntax for a quantified predicate:

```
expression relop { ALL | ANY | SOME } ( query_expression )
```

Example

```
10 < ANY
( SELECT COUNT(*)
    FROM order_tbl
    GROUP BY custid ;
)
```

BETWEEN Predicate

The BETWEEN predicate can be used to determine if a value is within a specified value range or not. The first expression specifies the lower bound of the range and the second expression specifies the upper bound of the range.

The predicate evaluates to true if the value is greater than or equal to the lower bound of the range and less than or equal to the upper bound of the range.

Syntax

This is the syntax for a BETWEEN predicate.

```
expression [ NOT ] BETWEEN expression AND expression
```

Example

```
salary BETWEEN 20000.00 AND 100000.00
```

NULL Predicate

The NULL predicate can be used for testing null values of database table columns.

Syntax

This is the syntax for a NULL predicate

```
column_name IS [ NOT ] NULL
```

Example

```
contact_name IS NOT NULL
```

LIKE Predicate

The LIKE predicate searches for strings that have a certain pattern. The pattern is specified after the LIKE keyword in a string constant. The pattern can be specified by a string in which the underscore (_) and percent sign (%) characters have special semantics.

Use the ESCAPE clause to disable the special semantics given to the characters (_) and

(%). The escape character specified must precede the special characters in order to disable their special semantics.

Syntax

This is the syntax for a LIKE predicate:

```
column_name [ NOT ] LIKE string_constant[ ESCAPE escape_character]
```

Notes

- The *column_name* specified in the LIKE predicate can be a column, a string constant, or an arbitrary character expression (such as SUBSTRING or LTRIM).
- The *string_constant* may be a string constant or a scalar function call.
- The *escape_character* must be a one character string constant.
- A percent sign (%) in the pattern matches zero or more characters of the column string.
- An underscore symbol (_) in the pattern matches any single character of the column string.
- The LIKE predicate is multi-byte enabled. The *string_constant* and the *escape_character* may contain multi-byte characters, and the *escape_character* can be a multi-byte character. A percent sign (%) or an underscore (_) in the *string_constant* can represent a multi-byte character. However, the percent sign or underscore itself must be the single-byte ASCII encoding.

Example

This example illustrates three ways to use the LIKE predicate:

```
cust_name LIKE '%Computer%'
 cust_name LIKE '___'
 item_name LIKE '%\_%' ESCAPE '\'
```

In the first LIKE clause, for all strings with the substring 'Computer' the predicate evaluates to true. In the second LIKE clause, for all strings which are exactly three characters long the predicate evaluates to true. In the third LIKE clause the backslash character (\ ) is specified as the escape character, which means that the special interpretation given to the underscore character (_) is disabled. The pattern evaluates to TRUE if the item_name column has embedded underscore characters.

EXISTS Predicate

The EXISTS predicate can be used to check for the existence of specific rows. The *query_expression* returns rows rather than values. The predicate evaluates to true if the number of rows returned by the *query_expression* is nonzero.

Syntax

This is the syntax for an EXISTS predicate:

```
EXISTS (query_expression)
```

Example

In this example, the predicate returns the item names of all items which contain sub-items:

```
SELECT itemname FROM pub.mtitem
  WHERE EXISTS(SELECT subitem FROM pub.mtitem);
```

IN Predicate

The IN predicate can be used to compare a value with a set of values. If an IN predicate specifies a query expression, then the result table it returns can contain only a single column.

Syntax

This is the syntax for an IN predicate:

```
expression [ NOT ] IN
 { (query_expression) | (constant , constant[ , ...] ) }
```

Example

```
address.state IN (‘MA', ‘NH')
```

OUTER JOIN Predicate

An outer join predicate specifies two tables and returns a result table that contains all the rows from one of the tables, even if there is no matching row in the other table.

Syntax

```
[ table_name. ] column  =  [ table_name . ] column  (+)
  | [ table_name.]column (+) = [table_name.]column
```

Numeric arithmetic expressions

Numeric arithmetic expressions compute a value using addition, subtraction, multiplication, and division operations on numeric literals and expressions that evaluate to any numeric data type.

Syntax

```
[ + | - ] { numeric_literal | numeric_expr }
 [ { + | - | * | / } numeric_arith_expr ]
```

Parameters

+ | -

Unary operators.

numeric_literal

Number value.

numeric_expr

Evaluates to a numeric data type:

+ | - | * | /

Operators for addition, subtraction, multiplication, and division. SQL evaluates numeric arithmetic expressions in the following order:

- Unary plus or minus
- Expressions in parentheses
- Multiplication and division, from left to right
- Addition and subtraction, from left to right

Date arithmetic expressions

Date arithmetic expressions compute the difference between date-time expressions in terms of days or milliseconds. SQL supports these forms of date arithmetic:

- Addition and subtraction of integers to and from date-time expressions
- Subtraction of one date-time expression from another

Syntax

```
date_time_expr { + | - }int_expr
  |date_time_expr - date_time_expr
```

Parameters

date_time_expr

Returns a value of type DATE or TIME or TIMESTAMP. A single date-time expression cannot mix data types, however. All elements of the expression must be the same data type.

Date-time expressions can contain date-time literals, but they must be converted to DATE or TIME using the CAST, CONVERT, or TO_DATE functions.

int_expr

Returns an integer value. SQL interprets the integer differently depending on the data type of the date-time expression:

- For DATE expressions, integers represent days
- For TIME expressions, integers represent milliseconds
- For TIMESTAMP expressions, integers represent milliseconds

Example

The following example manipulates DATE values using date arithmetic. SQL interprets integers as days and returns date differences in units of days:

```
SELECT C1, C2, C1-C2 FROM DTEST
c1                 c2           c1-c2
---------------------------------------
1956-05-07         1952-09-29  1316
   select sysdate,
     sysdate - 3 ,
     sysdate - cast ('9/29/52' as date)
from dtest;
 sysdate        sysdate-3    sysdate-convert(date,9/29/52)
-------------------------------------------------------
1995-03-24     1995-03-21     15516
```

The following example manipulates TIME values using date arithmetic. SQL interprets integers as milliseconds and returns time differences in milliseconds:

```
select systime,
     systime - 3000,
     systime - cast ('15:28:01' as time)
from dtest;
 systime      systime-3000   systime-convert(time,15:28:01)
-------------------------------------------------------
15:28:09     15:28:06       8000
```

8

Data type compatibility

This topic addresses compatibility issues when using the OpenEdge SQL environment and earlier versions of the Progress database. Specifically, it discusses mapping between Advanced Business Language (ABL) supported data types and the corresponding OpenEdge SQL data types.

For details, see the following topics:

- Supported ABL data types and corresponding OpenEdge SQL data types

Supported ABL data types and corresponding OpenEdge SQL data types

OpenEdge SQL supports many data types that do not correspond to ABL data types. The following table lists the ABL data types that **do** correspond to OpenEdge SQL data types.

Table 50: ABL an6d corresponding OpenEdge SQL data types

ABL data type	OpenEdge SQL data type
ARRAY	VARARRAY
BLOB	BLOB
CHARACTER	VARCHAR
CLOB	CLOB

ABL data type	OpenEdge SQL data type
DATE	DATE
DATETIME	TIMESTAMP
DATETIME-TZ	TIMESTAMP WITH TIME ZONE
DECIMAL	DECIMAL or NUMERIC
INTEGER	INTEGER
INT64	BIGINT
LOGICAL	BIT
RAW	VARBINARY
RECID	RECID

Notes

- All other SQL types are not compatible with ABL. In particular, OpenEdge SQL CHARACTER data is not compatible with the ABL. Use OpenEdge SQL type VARCHAR to map ABL CHARACTER data.
- Data columns created using OpenEdge SQL that have a data type not supported by ABL are not accessible through ABL applications and utilities.
- For more information about OpenEdge SQL data types, see OpenEdge SQL language elements on page 323.

9

OpenEdge SQL elements and statements in Backus Naur Form

This topic presents OpenEdge SQL language elements and statements in Backus Naur Form (BNF).

For details, see the following topics:

- Data types syntax in BNF

Data types syntax in BNF

Data Type

```
data_type ::=
char_data_type | exact_numeric_data_type | approx_numeric_data_type | date_time_data_type | bit_string_data_type
```

Character data type

```
char_data_type ::=
  { CHARACTER | CHAR } [ ( length ) ] |
  { CHARACTER VARYING | CHAR VARYING | CLOB | VARCHAR } [ (length) ]
```

Exact numeric data type

```
exact_numeric_data_type ::=
TINYINT
  | SMALLINT
  | INTEGER
  | NUMERIC | NUMBER [ ( precision [ , scale ] ) ]
  | DECIMAL [ ( precision , scale ) ]
```

Approximate numeric data type

```
approx_numeric_data_type ::=
  { REAL | DOUBLE PRECISION | FLOAT [ ( precision ) ] }
```

Date-time data type

```
date_time_data_type ::
DATE | TIME | TIMESTAMP | TIMESTAMP WITH TIME ZONE
```

Bit string data type

```
bit_string_data_type ::=
BIT | BINARY [ ( length ) ] | BLOB [ ( length ) ]
    | VARBINARY [( length ) ] | LONG VARBINARY [( length ) ]
```

Expressions syntax in BNF

Expression (expr)

```
expr ::=
 [{table_name.| alias.}]column_name
   |character_literal
   |numeric_literal
   |date-time_literal
   |aggregate_function
   |scalar_function
   |numeric_arith_expr
   |date_arith_expr
   |conditional_expr
   |  (expr)
```

Numeric arithmetic expression

```
numeric_arith_expr ::=
 [+| - ]{numeric_literal|numeric_expr}
 [{+| - | * |/}numeric_arith_expr]
```

Date arithmetic expression

```
date_arith_expr ::=
 date_time_expr{ + | - }int_expr|date_time_expr - date_time_expr
```

Conditional expression

Case expression

A type of conditional expression.

```
ase_expr ::=
 searched_case_exprsimple_case_expr
```

Searched case expression

```
searched_case_expr ::=
 CASE
  WHEN search_condition
  THEN {result_expr | NULL }[ , ...]
  [ ELSE expr | NULL ]
 END
```

Simple case expression

```
simple_case_expr ::=
  CASE primary_expr  WHEN expr THEN {result_expr | NULL }[ , ...]
  [ ELSE expr | NULL ]
  END
```

Literals syntax in BNF

Date literal

```
date-literal ::=
{ d'yyyy-mm-dd'}
   |mm-dd-yyyy | mm/dd/yyyy | mm-dd-yy | mm/dd/yy | yyyy-mm-dd
   | yyyy/mm/dd | dd-mon-yyyy | dd/mon/yyyy | dd-mon-yy | dd/mon/yy
```

Time literal

```
time_literal ::=
 { t 'hh:mi:ss'}|hh:mi:ss[:mls]
```

Timestamp literal

```
timestamp_literal ::=
{ t 'yyyy-mm-dd hh:mi:ss' } |'date_literal time_literal'
```

Timestamp with time zone literal

```
timestamp_with_time_zone_literal ::=
 { t 'yyyy-mm-dd hh:mi:ss - hh:mi:ss' } | 'date_literal time_literal'
```

Query Expressions syntax in BNF

Query expression

```
query_expression ::=
 query_specification  |  query_expressionset_operator
 query_expression  |  ( query_expression )
```

Set operator

```
set_operator ::={ UNION [ ALL ] | INTERSECT | MINUS }
```

Query specification

```
query_specification ::=
 SELECT [ ALL | DISTINCT ] [top_specification] { *
     | {table_name. | alias.}* [, {table_name. | alias.}* ] ,...
     | expr [ [ AS ] [ ' ]column_title[ ' ] ]
      [, expr [ [ AS ] [ ' ]column_title[ ' ] ] ] ,...}
 FROM table_ref[, table_ref] ...
   [ WHERE search_condition]
   [ GROUP BY [ table.]column_name[, [ table.]column_name] ,...
   [ HAVING search_condition] [WITH
locking_hints] [order_by_specification] [offset-fetch specification];
```

Table reference

```
table_ref ::=
 table_name[ AS ] [alias[ (column_alias[, ...] ) ] ]
    | (query_expression) [ AS ]alias
    [ (column_alias[ ,...] ) ] |
    [ ( ]joined_table[ ) ]
```

Joined table

```
joined_table ::=
 { table_ref CROSS JOIN table_ref
   | table_ref[ INNER | LEFT [ OUTER ]] JOIN
   table_ref ON search_condition}
```

From clause inner join

```
from_clause_inner_join ::=
{ FROM table_ref CROSS JOIN table_ref
  | FROM table_ref[ INNER ] JOIN table_ref
    ON search_condition}
```

Where clause inner join

```
where_clause_inner_join ::=
FROM table_ref, table_ref WHERE search_condition
```

From clause outer join

```
from_clause_outer_join ::=
FROM table_ref LEFT OUTER JOIN table_ref  ON search_condition
```

Where clause outer join

```
where_clause_outer_join ::=
 WHERE [table_name.]column (+) = [table_name.] column
  |WHERE [table_name.]column=[table_name.] column(+)
```

Search conditions syntax in BNF

Search condition

```
search_condition ::=
 [ NOT ]predicate[{ AND | OR }{predicate| ( search_condition ) }]
```

Predicate

```
predicate ::=
 basic_predicate| quantified_predicate| between_predicate| null_predicate
     | like_predicate| exists_predicate| in_predicate| outer_join_predicate
```

Relational operator

```
relop ::=
 =  | <> | != | ^= | < | <= | > | >=
```

Basic predicate

```
basic_predicate ::=
 expr relop { expr |( query_expression )}
```

Quantified predicate

```
quantified_predicate ::=
 exprrelop{ ALL | ANY | SOME } ( query_expression )
```

Between predicate

```
between_predicate ::=
 expr[ NOT ] BETWEEN expr AND expr
```

Null predicate

```
null_predicate ::=
 column_name IS [ NOT ] NULL
```

Like predicate

```
like_predicate ::=
 column_name[ NOT ] LIKE string_constant[ ESCAPE escape_character]
```

Exists predicate

```
exists_predicate ::=
  EXISTS (query_expression)
```

In predicate

```
in_predicate ::=
 expr[ NOT ] IN
  { (query_expression) | (constant , constant[ , ...] ) }
```

Outer join predicate

```
outer_join_predicate ::=
  [table_name.]column=[table_name.]column(+)
   | [table_name.]column(+) =[table_name.]column
```

Statements, DDL and DML syntax in BNF

This section lists OpenEdge SQL Data Definition Language (DDL) and Data Manipulation Language (DML) statements in Backus-Naur Form (BNF).

ALTER USER

```
alter user statement ::=
  ALTER USER { 'username' | 'username@domain_name' }, 'old_password',
'new_password';
```

CALL

```
call statement ::=
  CALL proc_name ( [parameter] [,...] ) ;
```

COMMIT

```
commit statement ::=
  COMMIT [ WORK ] ;
```

CREATE INDEX

```
create index statement ::=
  CREATE [ UNIQUE ] INDEX index_name
   ON table_name  ( {column_name[ ASC | DESC ]}[, ...] )
    [ AREA area_name] ;
```

CREATE PROCEDURE

```
create procedure statement ::=
CREATE PROCEDURE [owner_name.] procname
 ( [parameter_decl[ , ... ]] )
       [ RESULT ( column_namedata_type[ , ...] ) ]      [ IMPORT
     java_import_clause ]
    BEGIN
      java_snippet
    END
```

Parameter Declaration

```
parameter_decl ::= { IN | OUT | INOUT }parameter_namedata_type
```

CREATE SYNONYM

```
create synonym statement ::=
  CREATE [ PUBLIC ] SYNONYMsynonymFOR
    [ owner_name.]{table_name | view_name |synonym };
```

CREATE TABLE

```
create table statement ::=
 CREATE TABLE [owner_name.]table_name
  ( {column_definition |table_constraint}, . . . )
       [ AREA area_name ];

create table statement ::=
 CREATE TABLE [owner_name.]table_name
     [ (column_name[ NULL | NOT NULL ] , . . .) ]   [ AREA area_name]
      AS query_expression;
```

Column Definition

```
column_definition ::=
 column_namedata_type[ DEFAULT {literal| NULL | SYSDATE }]
 [column_constraint[column_constraint. . .]]
```

Column Constraint

```
column_constraint ::=
[ CONSTRAINT constraint_name]
   NOT NULL [ PRIMARY KEY | UNIQUE ]| REFERENCES
            [owner_name.]table_name[ ( column_name ) ]
       | CHECK ( search_condition )
```

Table Constraint

```
table_constraint ::=
 [ CONSTRAINT constraint_name ]
   PRIMARY KEY ( column [, . . . ] )
         | UNIQUE ( column [, . . . ] )
         | FOREIGN KEY ( column [, . . . ] )
      REFERENCES [owner_name.] table_name [ ( column [, . . . ] ) ]
         | CHECK ( search_condition )
```

CREATE TRIGGER

```
create trigger statement ::=
CREATE TRIGGER [owner_name.] trigname
  { BEFORE | AFTER }
  { INSERT | DELETE | UPDATE [ OF column_name [ , . . . ]}
    ON table_name [ REFERENCING { OLDROW [, NEWROW ] | NEWROW [, OLDROW ]}]
     [ FOR EACH { ROW | STATEMENT }][ IMPORT
        java_import_clause ]
     BEGIN
        java_snippet
     END
```

CREATE USER

```
create user statement ::=
CREATE USER {'username'|'username@domain_name'}, 'password' ;
```

CREATE VIEW

```
create view statement ::= CREATE VIEW [owner_name.] view_name [ ( column_name,
column_name, . . . ) ]  AS [ ( ] query_expression [ ) ][ WITH CHECK OPTION ] ;
```

DELETE

```
delete statement ::=
DELETE FROM [owner_name.] {table_name | view_name}[ WHERE search_condition ] ;
```

DROP INDEX

```
drop index statement ::=
 DROP INDEX  [ index_owner_name . ]  index_name
    [  ON  [ table_owner_name . ] table_name ]
```

DROP PROCEDURE

```
drop procedure statement ::=
DROP PROCEDURE [owner_name.]procedure_name ;
```

DROP SYNONYM

```
drop synonym statement ::=
DROP [ PUBLIC ] SYNONYM synonym ;
```

DROP TABLE

```
drop table statement ::=
DROP TABLE  [owner_name.] table_name ;
```

DROP TRIGGER

```
drop trigger statement ::=
DROP TRIGGER [owner_name.] trigger_name ;
```

DROP USER

```
drop user statement ::=
DROP USER {'username'|'username@domain_name'};
```

DROP VIEW

```
drop view statement ::=
DROP VIEW [owner_name.] view_name ;
```

GRANT RESOURCE, DBA

```
grant resource, dba statement ::=
GRANT { RESOURCE , DBA } TO user_name[ , user_name] , ...
;
```

GRANT PRIVILEGE

```
grant privilege statement::=
  GRANT {privilege[, privilege], ... | ALL [ PRIVILEGES]}
  ON table_name TO  {user_name[, user_name] , ... | PUBLIC }
    [WITH GRANT OPTION] ;
```

PRIVILEGE Syntax

```
privilege::=
 { SELECT | INSERT | DELETE | INDEX
  |UPDATE [ ( column , column , ... ) ]
  | REFERENCES [ ( column , column , ... ) ]}
```

INSERT

```
insert statement ::=
 INSERT INTO [owner_name.]{table_name|view_name}
  [ ( column_name[, column_name] , ... ) ]
    { VALUES ( value[, value] ,... ) |query_expression} ;
```

LOCK TABLE

```
lock table statement ::=
LOCK TABLE table_name [ , table_name ] , ...
 IN { SHARE | EXCLUSIVE } MODE ;
```

REVOKE RESOURCE, DBA

```
revoke resource, dba statement ::=
 REVOKE { RESOURCE | DBA } FROM {user_name [ , user_name ] ,...} ;
```

REVOKE PRIVILEGE

```
revoke privilege statement ::=
 REVOKE [ GRANT OPTION FOR ]
 {privilege[, privilege , ] , ... | ALL [ PRIVILEGES ]}
   ON table_name  FROM {user_name[ , user_name] , ... | PUBLIC }
   [ RESTRICT | CASCADE ] ;
```

PRIVILEGE Syntax

```
privilege ::=
 { SELECT | INSERT | DELETE | INDEX
   |UPDATE [ ( column , column , ... ) ]
   | REFERENCES [ ( column , column , ... ) ]}
```

ROLLBACK

```
rollback statement ::=
ROLLBACK [ WORK ] ;
```

SELECT

```
select statement ::=
query_expression  ORDER BY {expr|posn}[ ASC | DESC ]
     [ , {expr|posn}[ ASC | DESC ] , ...]
 FOR UPDATE [ OF [table.]column_name , ... ] ;
```

SET SCHEMA

```
set schema statement ::=
SET SCHEMA { 'string_literal' | ? | USER }
```

UPDATE

```
update statement ::=
 UPDATE table_name  SET assignment[,assignment], ...
  [ WHERE search_condition] ;
```

Assignment clause

```
assignment ::=
  column = {expr| NULL }| ( column[, column], ... ) = ( expr[, expr], ...
 )
   | ( column[, column], ... ) = ( query_expression )
```

UPDATE STATISTICS

```
update statistics statement ::=
UPDATE ([table_name|index_name|
  [ ALL ]column_name] STATISTICS [ AND ]) ...
  [ FOR table_name] ;
```

10

Compliance with industry standards

This topic identifies the level of ANSI SQL-92 compliance and ODBC SQL Grammar compliance for OpenEdge statements, and the SQL-92 and ODBC compatibility for OpenEdge SQL scalar functions.

For details, see the following topics:

- Scalar functions
- SQL-92 DDL and DML statements

Scalar functions

The following table lists OpenEdge SQL scalar functions. A check mark identifies the compatibility of the function as SQL-92 compatible, ODBC compatible, or a Progress® extension.

Table 51: Compatibility of SQL-92 scalar functions

Scalar function	SQL-92	ODBC	Progress extension	Notes
ABS	—	a	—	—
ACOS	—	a	—	—
ADD_MONTHS	—	—	a	—
ASCII	—	a	—	—

Scalar function	SQL-92	ODBC	Progress extension	Notes
ASIN	—	a	—	—
ATAN	—	a	—	—
ATAN2	—	a	—	—
CAST	a	—	—	—
CEILING	—	a	—	—
CHAR	—	a	—	—
CHR	—	—	a	—
COALESCE	a	—	—	—
CONCAT	—	a	—	—
CONVERT	—	a	—	Requires ODBC escape clause - { fn }
CONVERT	—	—	a	Not compatible with ODBC CONVERT
COS	—	a	—	—
CURDATE	—	a	—	—
CURTIME	—	a	—	—
DATABASE	—	a	—	—
DAYNAME	—	a	—	—
DAYOFMONTH	—	a	—	—
DAYOFWEEK	—	a	—	—
DAYOFYEAR	—	a	—	—
DB_NAME	—	—	a	—
DECODE	—	—	a	—
DEGREES	—	a	—	—
EXP	—	a	—	—
FLOOR	—	a	—	—

Scalar function	SQL-92	ODBC	Progress extension	Notes
GREATEST	—	—	a	—
HOUR	—	a	—	—
IFNULL	—	a	—	—
INITCAP	—	—	a	—
INSERT	—	a	—	—
INSTR	—	—	a	—
LAST_DAY	—	—	a	—
LCASE	—	a	—	—
LEAST	—	—	a	—
LEFT	—	a	—	—
LENGTH	—	a	—	—
LOCATE	—	a	—	—
LOG10	—	a	—	—
LOWER	a	—	—	—
LPAD	—	—	a	—
LTRIM	—	a	—	—
MINUTE	—	a	—	—
MOD	—	a	—	—
MONTH	—	a	—	—
MONTHNAME	—	a	—	—
MONTHS_BETWEEN	—	—	a	—
NEXT_DAY	—	—	a	—
NOW	—	a	—	—
NULLIF	a	—	—	—
NVL	—	—	a	—

Scalar function	SQL-92	ODBC	Progress extension	Notes
PI	—	a	—	—
POWER	—	—	a	—
PREFIX	—	—	a	—
PRO_ARR_DESCAPE	—	—	a	—
PRO_ARR_ESCAPE	—	—	a	—
PRO_ELEMENT	—	—	a	—
QUARTER	—	a	—	—
RADIAN	—	a	—	—
RAND	—	a	—	—
REPEAT	—	a	—	—
REPLACE	—	a	—	—
RIGHT	—	a	—	—
ROUND	—	—	a	—
ROWID	—	—	a	—
RPAD	—	—	a	—
RTRIM	—	a	—	—
SECOND	—	a	—	—
SIGN	—	a	—	—
SIN	—	a	—	—
SQRT	—	a	—	—
SUBSTR	—	—	a	—
SUBSTRING	—	a	—	—
SUFFIX	—	—	a	—
SYSDATE	—	—	a	—
SYSTIME	—	—	a	—

Scalar function	SQL-92	ODBC	Progress extension	Notes
SYSTIMESTAMP	—	—	a	—
TAN	—	a	—	—
TO_CHAR	—	—	a	—
TO_DATE	—	—	a	—
TO_NUMBER	—	—	a	—
TO_TIME	—	—	a	—
TO_TIMESTAMP	—	—	a	—
TRANSLATE	—	—	a	—
UCASE	—	a	—	—
UPPER	a	—	—	—
USER	a	a	a	—
WEEK	—	a	—	—
YEAR	—	a	—	—

SQL-92 DDL and DML statements

The following table lists OpenEdge SQL DDL and DML Statements. A check mark identifies the compliance of each statement as SQL-92, a level of ODBC SQL Grammar, or as a Progress extension.

Table 52: Compliance of SQL-92 DDL and DML statements

OpenEdge SQL statement	SQL-92	ODBC SQLgrammar	Progress extension	Notes
ALTER USER	—	—	a	—
CALL	—	Extended	—	Must enclose in an ODBC escape clause { fn }
COMMIT	a	—	—	—
CONNECT	a	—	USING *password*	—
CREATE INDEX	—	Core	AREA *area_name*	—

OpenEdge SQL statement	SQL-92	ODBC SQLgrammar	Progress extension	Notes
CREATE PROCEDURE	—	Core	a	—
CREATE SYNONYM	—	—	a	—
CREATE TABLE	a	Minimum	AREA AS *query_expression*	—
CREATE TRIGGER	—	Core	a	—
CREATE USER	—	—	a	—
CREATE VIEW	a	Core	—	—
DELETE	a	Extended	—	—
DISCONNECT	a	—	—	—
DROP INDEX	—	Core	—	—
DROP PROCEDURE	—	Core	a	—
DROP SYNONYM	—	—	a	—
DROP TABLE	—	Minimum	a	—
DROP TRIGGER	—	—	a	—
DROP USER	—	—	a	—
DROP VIEW	—	Core	a	—
GRANT	a	Core	INDEX RESOURCE DBA	—
INSERT	a	Core	—	—
LOCK TABLE	—	—	a	—
REVOKE	a	Core	INDEX RESOURCE DBA	—
ROLLBACK	a	—	—	—
SELECT	a	Extended	FOR UPDATE	—
SET CONNECTION	a	—	—	—
SET SCHEMA	a	—	—	—
SET TRANSACTION ISOLATION LEVEL	—	—	a	—

OpenEdge SQL statement	SQL-92	ODBC SQLgrammar	Progress extension	Notes
UPDATE	a	Extended	assignments of form: *(column, column)= (expr, expr)*	—
UPDATE STATISTICS	—	—	a	—

11

Syntax for ABL attributes

The OpenEdge SQL statements `CREATE TABLE` and `ALTER TABLE` allow you to define ABL (Advanced Business Language) attributes for tables and columns. This topic lists and describes the SQL keywords to use with `CREATE TABLE` and `ALTER TABLE` syntax. For examples of syntax using SQL keywords for ABL attributes, see the `CREATE TABLE` and `ALTER TABLE` entries in OpenEdge SQL statements on page 27.

For details, see the following topics:

- OpenEdge SQL keywords for ABL table attributes

OpenEdge SQL keywords for ABL table attributes

The following table lists the equivalent of ABL keywords to use while accessing ABL table attribute details with OpenEdge SQL statements.

Table 53: ABL table attributes used in OpenEdge SQL statements

Attribute keyword used in SQL statement	Description	Value
`PRO_DESCRIPTION`	Equivalent to ABL `DESCRIPTION.`	Arbitrary character string
`PRO_DUMP_NAME`	Equivalent to ABL `DUMP-NAME.`	Arbitrary character string

Attribute keyword used in SQL statement	Description	Value
PRO_FROZEN	Equivalent to ABL FROZEN. **Note:** OpenEdge SQL honors the value set here and does not allow modification of a frozen table using the ALTER TABLE, CREATE INDEX, CREATE PRO_WORD INDEX, DROP INDEX, or DROP TABLE commands. However, the frozen attribute may be set to 'N' to unfreeze a frozen table. For example: ALTER TABLE Customer SET PRO_FROZEN 'N';	'Y' | 'y' | 'N' | 'n'
PRO_HIDDEN	Indicates whether the table is shown in ABL tools and reports.	'Y' | 'y' | 'N' | 'n'
PRO_LABEL	Equivalent to ABL LABEL.	Arbitrary character string
PRO_VALEXP	Indicates an ABL validation expression.	Arbitrary character string
PRO_VALMSG	Indicates an ABL validation message.	Arbitrary character string
PRO_SA_VALMS	Indicates an ABL string attributes validation message.	Arbitrary character string
PRO_SA_LABEL	IndThe following tableicates an ABL table label.	Arbitrary character string
PRO_DEFAULT_INDEX	Determines default data-access index for a table.	Name of an index or table

The following table lists the keywords to use when setting ABL column attributes with OpenEdge SQL statements.

Table 54: ABL column attributes used in OpenEdge SQL statements

Attribute keyword used in SQL statement	Description	Value
PRO_COL_LABEL	Equivalent to ABL COL-LABEL	Free-form text

Attribute keyword used in SQL statement	Description	Value
PRO_DATA_TYPE	Equivalent to ABL DATA-TYPE **Note:** This attribute should be used only while converting INTEGER to INT64.	-
PRO_DESCRIPTION	Equivalent to ABL DESCRIPTION	Free-form text
PRO_FORMAT	Equivalent to ABL FORMAT	ABL format string
PRO_HELP	Indicates a ABL help message	Free-form text
PRO_LABEL	Indicates ABL label	Free-form text
PRO_RPOS	Indicates ABL relative record position	Positive integer
PRO_SQL_WIDTH	Indicates SQL width **Note:** This attribute works only with CHARACTER and Binary columns	Positive integer
PRO_VIEW_AS	Equivalent to ABL VIEW-AS	Free-form text
PRO_ORDER	Equivalent to ABL ORDER	Integer value
PRO_VALEXP	Indicates ABL validation expression	Free-form text
PRO_VALMSG	Indicates ABL validation message	Free-form text
PRO_SA_LABEL	Indicates ABL string attribute column label	Arbitrary character string
PRO_SA_COL_LABEL	Indicates ABL string attribute column label	Arbitrary character string
PRO_SA_FORMAT	Indicates ABL string attribute column label	Arbitrary character string
PRO_SA_INITIAL	Indicates ABL string attribute column label	Arbitrary character string
PRO_SA_HELP	Indicates ABL string attribute column label	Arbitrary character string

Attribute keyword used in SQL statement	Description	Value
PRO_SA_VALMSG	Indicates ABL string attribute column label	Arbitrary character string
DEFAULT **Note:** DEFAULT is a common attribute of both SQL and ABL.	Indicates the default value for a column.	A literal value whose type is compatible with the type of the column
PRO_CASE_SENSITIVE	Indicates case-sensitivity	'Y' | 'y' | 'N' | 'n'
PRO_LOB_SIZE_TEXT	The maximum size of a BLOB or CLOB column described as a string	Free-form text. For example: '32M'

The following table lists the keywords to use when setting ABL index attributes with OpenEdge SQL statements.

Table 55: ABL index attributes used in OpenEdge SQL statements

Attribute keyword used in SQL statement	Description	Value
PRO_ACTIVE	Changes the index's status from active to inactive. This action must be performed offline.	'n'
PRO_DESCRIPTION	Equivalent to ABL DESCRIPTION.	Free-form text

II

JDBC Reference

For details, see the following topics:

- Java class reference
- JDBC conformance notes

12

Java class reference

This topic provides information on OpenEdge™ SQL Java classes and methods.

For details, see the following topics:

- Java classes and methods
- DhSQLException
- DhSQLResultSet
- SQLCursor
- SQLIStatement
- SQLPStatement

Java classes and methods

This topic provides reference material on the OpenEdge SQL Java classes and methods. This section lists all the methods in the OpenEdge SQL Java classes and shows which classes declare them. Subsequent topics are arranged alphabetically and describe each class and its methods in more detail. Some Java methods are common to more than one class.

setParam

Sets the value of an SQL statement's input parameter to the specified value; a literal, procedure variable, or procedure input parameter. The following Java classes declare `setParam`:

- `SQLIStatement`

- SQLPStatement
- SQLCursor

makeNULL

Sets the value of an SQL statement's input parameter to NULL. The following Java classes declare makeNULL:

- SQLIStatement
- SQLPStatement
- SQLCursor

Sets a field of the currently active row in a procedure's result set to NULL:

- DhSQLResultSet

execute

Executes the SQL statement. The following Java classes declare execute:

- SQLIStatement
- SQLPStatement

rowCount

Returns the number of rows deleted, inserted, or updated by the SQL statement. The following Java classes declare rowCount:

- SQLIStatement
- SQLPStatement
- SQLCursor

open

Opens the result set specified by the SELECT or CALL statement. The following Java class declares open:

- SQLCursor

close

Closes the result set specified by the SELECT or CALL statement. The following Java class declares close:

- SQLCursor

fetch

Fetches the next record in a result set. The following Java class declares fetch:

- SQLCursor

found

Checks whether a fetch operation returned to a record. The following Java class declares `found`:

- `SQLCursor`

wasNULL

Checks if the value in a fetched field is `NULL`. The following Java class declares `wasNULL`:

- `SQLCursor`

getValue

Stores the value of a fetched field in the specified procedure variable or procedure output parameter. The following Java class declares `getValue`:

- `SQLCursor`

set

Sets the field in the currently active row of a procedure's result set a literal, procedure variable, or procedure input parameter. The following Java class declares `set`:

- `DhSQLResultSet`

insert

Inserts the currently active row into the result set of a procedure. The following Java class declares `insert`:

- `DhSQLResultSet`

getDiagnostics

Returns the specified detail of an error message. The following Java class declares `getDiagnostics`:

- `DhSQLException`

log

Writes a message to the log. The following Java classes inherit the log:

- `SQLIStatement`
- `SQLPStatement`
- `SQLCursor`
- `DhSQLResult Set`
- `DhSQLException`

err

Writes a message to the log. The following Java classes write to the log:

- SQLIStatement
- SQLPStatement
- SQLCursor
- DhSQLResult Set
- DhSQLException

DhSQLException

Extends the general java.lang. exception class to provide detail about errors in SQL statement execution. Any such errors raise an exception with an argument that is a SQLException class object. The getDiagnostics() method retrieves details of the error.

Constructors

```
public DhSQLException(int  ecode , String  errMsg )
```

Parameters

ecode

The error number associated with the exception condition.

errMsg

The error message associated with the exception condition.

In this example, the DhSQLException constructor creates an exception object called excep and then throws the excep object under all conditions:

```
CREATE PROCEDURE sp1_02()
BEGIN
     // raising exception
     DhSQLException excep = new DhSQLException(666,new String
          ("Entered the tst02 procedure"));
     if (true)
     throw excep;
END
```

DhSQLException.getDiagnostics

Returns the requested detail about an exception.

Format

`public String getDiagnostics(int` *`diagType`* `)`

Returns

A string containing the information specified by the *`diagType`* parameter, as shown in the table below.

Parameter

`diagType`

One of the argument values listed in the table below:

Table 56: Argument values for DhSQLException.getDiagnostics

Argument value	Returns
`RETURNED_SQLSTATE`	The `SQLSTATE` returned by execution of the previous SQL statement
`MESSAGE_TEXT`	The condition indicated by `RETURNED_SQLSTATE`
`CLASS_ORIGIN`	Not currently used; always returns `NULL`
`SUBCLASS_ORIGIN`	Not currently used; always returns `NULL`

Throws

`DhSQLException`

Example

This code fragment illustrates `DhSQLException.getDiagnostics`:

```
try
{     SQLIStatement insert_cust = new SQLIStatement (
     "INSERT INTO customer VALUES (1,2) ");
}catch (DhSQLException e)
{     errstate = e.getDiagnostics (RETURNED_SQLSTATE) ;
     errmesg  = e.getDiagnostics (MESSAGE_TEXT) ;
          .
          .
          .
}
```

DhSQLException.err

The DhSQLException.err method can be used inside Java Stored Procedures and Java Database SQL Triggers. On invoking this method, error messages are written to the sqlNNNN.trc file. The sqlNNNN.trc file is created in the working directory and NNNN stands for the SQL server process PID. The following is an example of an error message:

```
DROP PROCEDURE Fix;
CREATE PROCEDURE Fix ()
   IMPORT
      import java.sql.*;
   BEGIN
      if (true) {
      DhSQLException excep = new DhSQLException ( 666 , new String("Sample user defined
 new DhSQLException object") );
      excep.err(excep.getDiagnostics(DhSQLException.RETURNED_SQLSTATE));
      excep.err("\r");
      excep.err(excep.getDiagnostics(DhSQLException.MESSAGE_TEXT));
      excep.err("\r");
      }
   END;
COMMIT WORK;
```

DhSQLResultSet

Provides the stored procedure with a result set to return to the application that called the procedure.

The Java code in a stored procedure does not explicitly create `DhSQLResultSet` objects. Instead, when the SQL server creates a Java class from a `CREATE PROCEDURE` statement that contains a Result clause, it implicitly instantiates an object of type `DhSQLResultSet`, and calls it `SQLResultSet`.

Procedures invoke methods of the `SQLResultSet` instance to populate fields and rows of the result set.

Constructors

No explicit constructor

Parameters

None

Throws

`DhSQLException`

DhSQLResultSet.insert

Inserts the currently active row into a procedure's result set.

Format

```
public void insert()
```

Returns

None

Parameters

None

Throws

`DhSQLException`

Example

This code fragment illustrates `SQLResultSet.set` and `SQLResultSet.insert`:

```
CREATE PROCEDURE get_sal2 ()
RESULT (
    empname CHAR(20),
    empsal   NUMERIC,
)BEGIN
    String ename = new String (20) ;
    BigDecimal esal = new BigDecimal () ;
    SQLCursor empcursor = new SQLCursor (
        "SELECT name, sal FROM emp " ) ;

    empcursor.Open () ;
    do
    {
        empcursor.Fetch ();
        if (empcursor.found ())
        {
                empcursor.getValue (1, ename);
                empcursor.getValue (2, esal);
            SQLResultSet.Set (1, ename);
            SQLResultSet.Set (2, esal);
            SQLResultSet.Insert ();
        }
    } while (empcursor.found ()) ;
    empcursor.close () ;
END
```

DhSQLResultSet.makeNULL

Sets a field of the currently active row in a procedure's result set to `NULL`. This method is redundant with using the `DhSQLResultSet.set` method to set a procedure result-set field to `NULL`.

Format

`public void makeNULL(int` *`field`* `)`

Returns

None

Parameter

field

An integer that specifies which field of the result-set row to set to NULL. 1 denotes the first field in the row, 2 denotes the second, *n* denotes the *nth*.

Throws

DhSQLException

Example

This code fragment illustrates SQLResultSet.set and SQLResultSet.makeNULL:

```
CREATE PROCEDURE test_makeNULL2(
    IN char_in CHAR(20)
RESULT ( res_char CHAR(20) , res_vchar VARCHAR(30))
 BEGIN
    SQLResultSet.set(1,char_in);
    SQLResultSet.makeNULL(2);
END
```

DhSQLResultSet.set

Sets the field in the currently active row of a procedure's result set to the specified value (a literal, procedure variable, or procedure input parameter).

Format

```
public void set(int  field , Object  val )
```

Returns

None

Parameters

field

An integer that specifies which field of the result-set row to set to the value specified by val. (1 denotes the first field in the row, 2 denotes the second, and so on.)

val

A literal or the name of a variable or input parameter that contains the value to be assigned to the field.

Throws

DhSQLException

Example

This code fragment illustrates SQLResultSet.Set:

```
CREATE PROCEDURE get_sal2 ()
RESULT (
     empname CHAR(20),
     empsal   NUMERIC,
)BEGIN
     String ename = new String (20) ;
     BigDecimal esal = new BigDecimal () ;
     SQLCursor empcursor = new SQLCursor (
        "SELECT name, sal FROM emp " ) ;

     empcursor.Open () ;
     do
     {
        empcursor.Fetch ();
        if (empcursor.found ())
          {
               empcursor.getValue (1, ename);
               empcursor.getValue (2, esal);
               SQLResultSet.Set (1, ename);
               SQLResultSet.Set (2, esal);
               SQLResultSet.Insert ();
          }
     } while (empcursor.found ()) ;
     empcursor.close () ;
END
```

SQLCursor

Allows rows of data to be retrieved from a database or another stored procedure's result set.

Constructors

SQLCursor (String *statement*)

Parameters

statement

Generates a result set. Enclose the SQL statement in double quotes. The SQL statement is either a SELECT or CALL statement.

Notes

- A SELECT statement queries the database and returns data that meets the criteria specified by the query expression in the SELECT statement.
- A CALL statement invokes another stored procedure that returns a result set specified by the RESULT clause of the CREATE PROCEDURE statement.

Throws

DhSQLException

Example

The following excerpt from a stored procedure instantiates an SQLCursor object called cust_cursor that retrieves data from a database table:

```
SQLCursor empcursor = new SQLCursor ( "SELECT name, sal FROM emp " ) ;
```

The following excerpt from a stored procedure instantiates an SQLCursor object called cust_cursor that calls another stored procedure:

```
t_cursor = new SQLCursor ( "CALL get_customers (?) ") ;
```

SQLCursor.close

Closes the result set specified by a SELECT or CALL statement.

Format

```
public void close()
```

Returns

None

Parameters

None

Throws

DhSQLException

Example

This code fragment illustrates the getValue and close methods:

```
{
     if (cust_cursor.Found ())
     {
          cust_cursor.getValue (1, cust_number);
          cust_cursor.getValue (2, cust_name) ;
     }
     else
          break;
}     cust_cursor.close () ;
```

SQLCursor.fetch

Fetches the next record in a result set, if there is one.

Format

```
public void fetch()
```

Returns

None

Parameters

None

Throws

`DhSQLException`

Example

This code fragment illustrates the `fetch` method and the `getValue` method:

```
for (;;)
{     cust_cursor.Fetch ();
     if (cust_cursor.Found ())
     {
          cust_cursor.getValue (1, cust_number);
          cust_cursor.getValue (2, cust_name) ;
     }
     else
          break;
}
```

SQLCursor.found

Checks whether a `fetch` operation returned a record.

Format

```
public boolean found ()
```

Returns

True if the previous call to fetch() returned a record, false otherwise

Parameters

None

Throws

`DhSQLException`

Example

This code fragment illustrates the `fetch`, `found`, and `getValue` methods:

```
for (;;)
{     cust_cursor.Fetch ();
     if (cust_cursor.Found ())
     {
          cust_cursor.getValue (1, cust_number);
          cust_cursor.getValue (2, cust_name) ;
     }
     else
          break;
}
```

SQLCursor.getParam

Retrieves the values of Java `OUT` and `INOUT` parameters.

Format

```
inout_var = getParam( int  fieldIndex , short  fieldType  );
```

Returns

`OUT` or `INOUT` variable

Parameters

`inout_var`

The target variable into which the value of an `OUT` or `INOUT` parameter is stored.

`fieldIndex`

An integer that specifies the position of the parameter in the parameter list.

`fieldType`

A short integer that specifies the data type of the parameter. The allowable defined values for *`fieldType`* are listed in the table below, grouped by category of data type.

Table 57: Allowable values for fieldType in getParam

Character	Exact numeric	Approximate numeric	Date-time	Bit string
CHAR	INTEGER	REAL	DATE	BIT
CHARACTER	SMALLINT	FLOAT	TIME	BINARY
VARCHAR	TINYINT	DOUBLE	TIMESTAMP	VARBINARY

Character	Exact numeric	Approximate numeric	Date-time	Bit string
—	NUMERIC	—	—	LVARBINARY
—	DECIMAL	—	—	—

Throws

DhSQLException

Notes

- The getParam() method returns the value of an INOUT or OUT parameter identified by the number you specify in the *fieldIndex*parameter. getParam() returns the value as an object of the data type you specify in the *fieldType*parameter. Since getParam() returns the result as an instance of class Object, you must explicitly cast your *inout_var* variable to the correct data type.
- If the OUT or INOUT parameter is of data type CHARACTER, then getParam returns a Java String Object. You must declare a procedure variable of type String, and explicitly cast the value returned by getParam to type String. Before calling getParam() you must call the SQLCursor.wasNULL method to test whether the returned value is NULL. If getParam() is called for a NULL value, it raises a DhSQLException.

SQLCursor.getValue

Assigns a single value from an SQL result set to a procedure variable. The single field value is the result of an SQL query or the result from another stored procedure.

Format

```
public Object getValue( int  fieldNum , short  fieldType  )
```

Returns

Object

Parameters

fieldNum

An integer that specifies the position of the field to retrieve from the fetched record.

fieldType

A short integer that specifies the data type of the parameter. The allowable defined values for *fieldType* are listed in the table below, grouped by category of data type.

Table 58: Allowable values for fieldType in getValue

Character	Exact numeric	Approximate numeric	Date-time	Bit string
CHAR	INTEGER	REAL	DATE	BIT

Character	Exact numeric	Approximate numeric	Date-time	Bit string
CHARACTER	SMALLINT	FLOAT	TIME	BINARY
VARCHAR	TINYINT	DOUBLE	TIMESTAMP	VARBINARY
—	NUMERIC	—	—	LVARBINARY
—	DECIMAL	—	—	—

Throws

DhSQLException

Notes

- Before invoking getValue, you must test for the NULL condition by calling the SQLCursor.wasNULL method. If the value returned is NULL, you must explicitly set the target variable in the stored procedure to NULL.
- The getValue method returns a value from the result set identified by the number you specify in the *fieldNum* parameter. getValue returns the value as an object of the data type you specify in the *fieldType* parameter. Since getValue returns the result as an instance of class Object, you must explicitly cast your return value to the correct data type.
- If the returned value is of data type CHARACTER, then getValue returns a Java String Object. You must declare a procedure variable of type String and explicitly cast the value returned by getValue to type String.

Example

This example illustrates testing for NULL and invoking the Java getValue method:

```
Integer  pvar_int = new Integer(0);
String   pvar_str = new String();
SQLCursor select_t1 = new SQLCursor
     ("select int_col, char_col from T1");
 Select_t1.open();
Select_t1.fetch();
 while(select_t1.found())
{     // Assign values from the current row of the SQL result set
     // to the procedure variables. First check whether
     // the values fetched are null. If null then explicitly
     // set the procedure variables to null.
      if ((select_t1.wasNULL(1)) == true)
           pvar_int = null;
     else
           pvar_int = (Integer)select_t1.getValue(1, INTEGER);
     if ((select_t1.wasNULL(2)) == true)
           pvar_str = null;
     else
           pvar_str = (String)select_t1.getValue(1, CHAR);
}
```

SQLCursor.makeNULL

Sets the value of an SQL statement's input parameter to NULL. This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes. This method is redundant with using the setParam method to set an SQL statement's input parameter to NULL.

Format

```
public void makeNULL(int  f )
```

Returns

None

Parameter

f

An integer that specifies which input parameter of the SQL statement string to set to NULL. 1 denotes the first input parameter in the statement, 2 denotes the second, *n* denotes the *nth*.

Throws

DhSQLException

Example

This code fragment illustrates the makeNULL method:

```
CREATE PROCEDURE sc_makeNULL()
BEGIN
     SQLCursor select_btypes = new SQLCursor (
         "SELECT small_fld from sfns where small_fld = ? ");
     select_btypes.makeNULL(1);
     select_btypes.open();
     select_btypes.fetch();
   .
   .
   .
     select_btypes.close();
END
```

SQLCursor.open

Opens the result set specified by the SELECT or CALL statement.

Format

```
public void open()
```

Returns

None

Parameters

None

Throws

`DhSQLException`

Example

This code fragment illustrates the `open` method:

```
SQLCursor empcursor = new SQLCursor ( "SELECT name, sal FROM emp " ) ;
empcursor.Open () ;
```

SQLCursor.registerOutParam

Registers `OUT` parameters.

Format

```
registerOutParam( int fieldIndex , short fieldType [ , short scale ] )
```

Returns

None

Parameters

`fieldIndex`

An integer that specifies the position of the parameter in the parameter list.

`fieldType`

A short integer that specifies the data type of the parameter.

The allowable defined values for *`fieldType`* are listed in the table below, grouped by category of data type.

Table 59: Allowable values for fieldType in registerOutParam

Character	Exact numeric	Approximate numeric	Date-time	Bit string
CHAR	INTEGER	REAL	DATE	BIT
CHARACTER	SMALLINT	FLOAT	TIME	BINARY
VARCHAR	TINYINT	DOUBLE	TIMESTAMP	VARBINARY

Character	Exact numeric	Approximate numeric	Date-time	Bit string
—	NUMERIC	—	—	LVARBINARY
—	DECIMAL	—	—	—

Throws

DhSQLException

SQLCursor.rowCount

Returns the number of rows affected (selected, inserted, updated, or deleted) by the SQL statement. This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes.

Format

```
public int rowCount()
```

Returns

An integer indicating the number of rows.

Parameters

None

Throws

DhSQLException

Example

This example uses the rowCount method of the SQLIStatement class by nesting the method invocation within SQLResultSet.set to store the number of rows affected (1, in this case) in the procedure's result set:

```
CREATE PROCEDURE sis_rowCount()
RESULT ( ins_recs INTEGER )
 BEGIN
     SQLCursor insert_test103 = new SQLIStatement (
     "INSERT INTO test103 (fld1) values (17)");
     insert_test103.execute();
     SQLResultSet.set(1,new Long(insert_test103.rowCount()));
     SQLResultSet.insert();
END
```

SQLCursor.setParam

Sets the value of an SQL statement's input parameter to the specified value (a literal, procedure variable, or procedure input parameter). This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes.

Format

```
public void setParam(int  f , Object  val )
```

Returns

None

Parameters

f

An integer that specifies which parameter marker in the SQL statement is to receive the value. 1 denotes the first parameter marker, 2 denotes the second, *n*denotes the *nth*.

val

A literal or the name of a variable or input parameter that contains the value to be assigned to the parameter marker.

Throws

DhSQLException

Example

This code fragment illustrates the setParam method:

```
CREATE PROCEDURE sps_setParam()
 BEGIN
// Assign local variables to be used as SQL input parameter references
     Integer ins_fld_ref   = new Integer(1);
     Integer ins_small_fld = new Integer(3200);
     Integer ins_int_fld   = new Integer(21474);
     Double  ins_doub_fld  = new Double(1.797E+30);
     String  ins_char_fld  = new String("Athula");
     String  ins_vchar_fld = new String("Scientist");
     Float   ins_real_fld  = new Float(17);
     SQLIStatement insert_sfns1 = new SQLIStatement ("INSERT INTO sfns
      (fld_ref,small_fld,int_fld,doub_fld,char_fld,vchar_fld)
      values (?,?,?,?,?,?)" );
     insert_sfns1.setParam(1,ins_fld_ref);
     insert_sfns1.setParam(2,ins_small_fld);
     insert_sfns1.setParam(3,ins_int_fld);
     insert_sfns1.setParam(4,ins_doub_fld);
     insert_sfns1.setParam(5,ins_char_fld);
     insert_sfns1.setParam(6,ins_vchar_fld);
     insert_sfns1.execute();
END
```

SQLCursor.wasNULL

Checks if the value in a fetched field is NULL.

Format

`public boolean wasNULL(int` *`field`* `)`

Returns

True if the field is `NULL`, false otherwise

Parameter

`field`

An integer that specifies which field of the fetched record is of interest. (1 denotes the first column of the result set, 2 denotes the second, and so on.) `wasNULL` checks whether the value in the currently fetched record of the column denoted by field is `NULL`.

Throws

`DhSQLException`

Example

This code fragment illustrates the `wasNULL` method:

```
CREATE PROCEDURE test_wasNULL()
 BEGIN
     int small_sp      = 0;
     SQLCursor select_btypes =
          new SQLCursor ("SELECT small_fld from sfns");
     select_btypes.open();
     select_btypes.fetch();
     if ((select_btypes.wasNULL(1)) == true)
            small_sp = null;
     else
            select_btypes.getValue(1,small_sp);
     select_btypes.close();
END
```

SQLIStatement

Allows immediate (one-time) execution of SQL statements that do not generate a result set.

Constructors

`SQLIStatement` (`String`*`statement`*)

Parameter

`statement`

An SQL statement that does not generate a result set. Enclose the SQL statement in double quotes.

Throws

DhSQLException

Example

This code fragment illustrates the SQLIStatement class:

```
CREATE PROCEDURE insert_customer (
IN  cust_number INTEGER,
IN  cust_name   CHAR(20)
) BEGIN
     SQLIStatement insert_cust = new SQLIStatement (
        "INSERT INTO customer VALUES (?,?) ");
END
```

SQLIStatement.execute

Executes the SQL statement. This method is common to the SQLIStatement and SQLPStatement classes.

Format

```
public void execute()
```

Returns

None

Parameters

None

Throws

DhSQLException

Example

This code fragment illustrates the setParam and execute methods:

```
CREATE PROCEDURE insert_customer (
IN  cust_number INTEGER,
IN  cust_name   CHAR(20)
) BEGIN
     SQLIStatement insert_cust = new SQLIStatement (
            "INSERT INTO customer VALUES (?,?) ");
     insert_cust.setParam (1, cust_number);
     insert_cust.setParam (2, cust_name);
     insert_cust.execute ();
END
```

SQLIStatement.makeNULL

Sets the value of an SQL statement's input parameter to NULL. This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes. This method is redundant with using the setParam method to set an SQL statement's input parameter to NULL.

Format

```
public void makeNULL(int  f )
```

Returns

None

Parameter

f

An integer that specifies which input parameter of the SQL statement string to set to NULL. 1 denotes the first input parameter in the statement, 2 denotes the second, *n* denotes the *nth*.

Throws

DhSQLException

Example

This code fragment illustrates the makeNULL method:

```
CREATE PROCEDURE sis_makeNULL()
 BEGIN
     SQLIStatement insert_sfns1 = new SQLIStatement ("INSERT INTO sfns
      (fld_ref,small_fld,int_fld,doub_fld,char_fld,vchar_fld)
      values (?,?,?,?,?,?)" );
     insert_sfns1.setParam(1,new Integer(66));
     insert_sfns1.makeNULL(2);
     insert_sfns1.makeNULL(3);
     insert_sfns1.makeNULL(4);
     insert_sfns1.makeNULL(5);
     insert_sfns1.makeNULL(6);
     insert_sfns1.execute();
END
```

SQLIStatement.rowCount

Returns the number of rows affected (selected, inserted, updated, or deleted) by the SQL statement. This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes.

Format

```
public int rowCount()
```

Returns

An integer indicating the number of rows

Parameters

None

Throws

`DhSQLException`

Example

This example uses the `rowCount` method of the `SQLIStatement` class by nesting the method invocation within `SQLResultSet.set` to store the number of rows affected (`1`, in this case) in the procedure's result set:

```
CREATE PROCEDURE sis_rowCount()
RESULT ( ins_recs INTEGER )
BEGIN
     SQLIStatement insert_test103 = new SQLIStatement (
     "INSERT INTO test103 (fld1) values (17)");
     insert_test103.execute();
     SQLResultSet.set(1,new Long(insert_test103.rowCount()));
     SQLResultSet.insert();
END
```

SQLIStatement.setParam

Sets the value of an SQL statement's input parameter to the specified value (a literal, procedure variable, or procedure input parameter). This method is common to the `SQLCursor`, `SQLIStatement`, and `SQLPStatement` classes.

Format

```
public void setParam(int  f , Object  val )
```

Returns

None

Parameters

f

An integer that specifies which parameter marker in the SQL statement is to receive the value (`1` denotes the first parameter marker, `2` denotes the second, and so on).

val

A literal or the name of a variable or input parameter that contains the value to be assigned to the parameter marker.

Throws

`DhSQLException`

Example

This code fragment illustrates the `setParam` method:

```
CREATE PROCEDURE sps_setParam()
 BEGIN
// Assign local variables to be used as
// SQL input parameter references
     Integer ins_fld_ref   = new Integer(1);
     Integer ins_small_fld = new Integer(3200);
     Integer ins_int_fld   = new Integer(21474);
     Double  ins_doub_fld  = new Double(1.797E+30);
     String  ins_char_fld  = new String("Athula");
     String  ins_vchar_fld = new String("Scientist");
     Float   ins_real_fld  = new Float(17);
      SQLPStatement insert_sfns1 = new SQLPStatement ("INSERT INTO sfns
      (fld_ref,small_fld,int_fld,doub_fld,char_fld,vchar_fld)
      values (?,?,?,?,?,?)" );
      insert_sfns1.setParam(1,ins_fld_ref);
     insert_sfns1.setParam(2,ins_small_fld);
     insert_sfns1.setParam(3,ins_int_fld);
     insert_sfns1.setParam(4,ins_doub_fld);
     insert_sfns1.setParam(5,ins_char_fld);
     insert_sfns1.setParam(6,ins_vchar_fld);
     insert_sfns1.execute();
END
```

SQLPStatement

Allows prepared (repeated) execution of SQL statements that do not generate a result set.

Constructors

`SQLPStatement` (`String`*`statement`*)

Parameter

`statement`

An SQL statement that does not generate a result set. Enclose the SQL statement in double quotes.

Throws

`DhSQLException`

Example

This code fragment illustrates the SQLPStatement class:

```
SQLPStatement pstmt = new SQLPStatement ( "INSERT INTO T1 VALUES (?, ?) ") ;
```

SQLPStatement.execute

Executes the SQL statement. This method is common to the SQLIStatement and SQLPStatement classes.

Format

```
public void execute()
```

Returns

None

Parameters

None

Throws

DhSQLException

Example

This code fragment illustrates the execute and setParam methods in the SQLPStatement class:

```
SQLPStatement pstmt = new SQLPStatement (
     "INSERT INTO T1 VALUES (?, ?) ") ;
pstmt.setParam (1, 10);
pstmt.setParam (2, 10);
pstmt.execute ();
pstmt.setParam (1, 20);
pstmt.setParam (2, 20);
pstmt.execute ();
```

SQLPStatement.makeNULL

Sets the value of an SQL statement's input parameter to NULL. This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes. This method is redundant with using the setParam method to set an SQL statement's input parameter to NULL.

Format

```
public void makeNULL(int f )
```

Returns

None

Parameter

f

An integer that specifies which input parameter of the SQL statement string to set to NULL. (1 denotes the first input parameter in the statement, 2 denotes the second, and so on.)

Throws

DhSQLException

Example

This code fragment illustrates SQLPStatement.makeNULL:

```
CREATE PROCEDURE sps_makeNULL()
BEGIN
     SQLPStatement insert_sfns1 = new SQLPStatement ("INSERT INTO sfns
      (fld_ref,small_fld,int_fld,doub_fld,char_fld,vchar_fld)
      values (?,?,?,?,?,?)" );
     insert_sfns1.setParam(1,new Integer(666));
     insert_sfns1.makeNULL(2);
     insert_sfns1.makeNULL(3);
     insert_sfns1.makeNULL(4);
     insert_sfns1.makeNULL(5);
     insert_sfns1.makeNULL(6);
     insert_sfns1.execute();
END
```

SQLPStatement.rowCount

Returns the number of rows affected (selected, inserted, updated, or deleted) by the SQL statement. This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes.

Format

```
public int rowCount()
```

Returns

An integer indicating the number of rows

Parameters

None

Throws

DhSQLException

Example

This example uses the rowCount method of the SQLPStatement class by nesting the method invocation within SQLResultSet.set to store the number of rows affected (1, in this case) in the procedure's result set:

```
CREATE PROCEDURE sis_rowCount()
RESULT ( ins_recs INTEGER )
BEGIN
     SQLPStatement insert_test103 = new SQLPStatement (
     "INSERT INTO test103 (fld1) values (17)");
     insert_test103.execute();
     SQLResultSet.set(1,new Long(insert_test103.rowCount()));
     SQLResultSet.insert();
END
```

SQLPStatement.setParam

Sets the value of an SQL statement's input parameter to the specified value (a literal, procedure variable, or procedure input parameter). This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes.

Format

```
public void setParam(int  f , Object  val )
```

Returns

None

Parameters

f

An integer that specifies which parameter marker in the SQL statement is to receive the value (1 denotes the first parameter marker, 2 denotes the second, and so on).

val

A literal or the name of a variable or input parameter that contains the value to be assigned to the parameter marker.

Throws

DhSQLException

Example

This code fragment illustrates SQLPStatement.setParam:

```
CREATE PROCEDURE sps_setParam()
BEGIN
// Assign local variables to be used as
// SQL input parameter references
     Integer ins_fld_ref   = new Integer(1);
     Integer ins_small_fld = new Integer(3200);
     Integer ins_int_fld   = new Integer(21474);
     Double  ins_doub_fld  = new Double(1.797E+30);
     String  ins_char_fld  = new String("Athula");
     String  ins_vchar_fld = new String("Scientist");
     Float   ins_real_fld  = new Float(17);
     SQLPStatement insert_sfns1 = new SQLPStatement ("INSERT INTO sfns
      (fld_ref,small_fld,int_fld,doub_fld,char_fld,vchar_fld)
      values (?,?,?,?,?,?)" );
      insert_sfns1.setParam(1,ins_fld_ref);
     insert_sfns1.setParam(2,ins_small_fld);
     insert_sfns1.setParam(3,ins_int_fld);
     insert_sfns1.setParam(4,ins_doub_fld);
     insert_sfns1.setParam(5,ins_char_fld);
     insert_sfns1.setParam(6,ins_vchar_fld);
     insert_sfns1.execute();
END
```

13

JDBC conformance notes

This topic details the DataDirect JDBC driver's support for the JDBC standard.

For details, see the following topics:

- Supported data types
- Return values for DatabaseMetaData

Supported data types

The Data Direct JDBC Driver supports standard JDBC mapping of JDBC data types to corresponding Java data types.

In the JDBC methods `CallableStatement.getXXX` and `PreparedStatement.setXXX` methods, `XXX` is a Java type:

- For `setXXX` methods, the driver converts the Java data type to the JDBC data type shown in the table below before sending it to the database.

  The following table provides details on data type mapping between Java and JDBC data types.

  Table 60: Mapping between Java and JDBC data types

Java data type	JDBC data type
Boolean	`BIT`
`Byte`	`TINYINT`

Java data type	JDBC data type
byte [ ]	LONGVARBINARY
byte [ ]	VARBINARY
double	DOUBLE
float	REAL
Int	INTEGER
java.math.BigDecimal	NUMERIC, DECIMAL
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
Short	SMALLINT
String	VARCHAR

- For getXXX methods, the driver converts the JDBC data type returned by the database to the Java data type (shown in the tables below) before returning it to the getXXX method.

  The following table details mapping between JDBC and Java data types.

  Table 61: Mapping between JDBC and Java data types

JDBC data type	Java data type
BIT	boolean
CHAR	String
DECIMAL	java.math.BigDecimal
INTEGER	int
NUMERIC	java.math.BigDecimal
SMALLINT	short
TINYINT	byte
VARCHAR	String

  The following table details mapping between SQL-92 and Java data types.

Table 62: Mapping between SQL-92 and Java data types

SQL-92 data type	Java data type
BINARY	byte[ ]
BIT	boolean
CHAR, VARCHAR	String
DATE	java.sql.Date
DECIMAL	java.math.BigDecimal
DOUBLE PRECISION	Double
FLOAT	Float
INTEGER	Integer
LONGVARBINARY	byte[ ]
NUMERIC	java.math.BigDecimal
REAL	Float
SMALLINT	short
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
TINYINT	byte[ ]
VARBINARY	byte[ ]

The following table provides information on JDBC data type conversion.

Table 63: JDBC data type conversion

JDBC data type	Converts to . . .
BIGINT	CHAR, DOUBLE, FLOAT, INTEGER, SMALLINT, TINYINT
BINARY	Does not convert to any other data type
BIT	Does not convert to any other data type
CHAR	BIGINT, DATE. DECIMAL, DOUBLE, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TIME, TIMESTAMP, TINYINT, VARCHAR

JDBC data type	Converts to . . .
DATE	CHAR, TIMESTAMP, VARCHAR
DECIMAL	BIGINT, CHAR, DOUBLE, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT, VARCHAR
DOUBLE	BIGINT, CHAR, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT, VARCHAR
FLOAT	BIGINT, CHAR, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT, VARCHAR
INTEGER	BIGINT, CHAR, DECIMAL, DOUBLE, FLOAT, NUMERIC, REAL, SMALLINT, TINYINT, VARCHAR
LONGVARBINARY	Does not convert to any other data type
LONGVARCHAR	Does not convert to any other data type
NUMERIC	BIGINT, CHAR, DECIMAL, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINYINT, VARCHAR
REAL	BIGINT, CHAR, DECIMAL, DOUBLE, FLOAT, INTEGER, NUMERIC, SMALLINT, TINYINT, VARCHAR
SMALLINT	BIGINT, CHAR, DECIMAL, DOUBLE, FLOAT, INTEGER, NUMERIC, REAL, TINYINT, VARCHAR
TIME	CHAR, TIMESTAMP
TIMESTAMP	CHAR, DATE, TIME, VARCHAR
TINYINT	BIGINT, CHAR, DECIMAL, DOUBLE, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, VARCHAR
VARBINARY	Does not convert to any other data type
VARCHAR	BIGINT, CHAR, DATE, DECIMAL, DOUBLE, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TIME, TIMESTAMP, TINYINT

Return values for DatabaseMetaData

Applications call methods of the DatabaseMetaData class to retrieve details about the JDBC support provided by the OpenEdge™ SQL JDBC driver.

The following table lists each method of the DatabaseMetaData class and shows what the JDBC driver returns when an application calls the method. For details on the format and usage of each method, see the Java Core API documentation for your platform. Many of the methods return lists of information as an object of type `ResultSet`. Use the normal `ResultSet` methods, such as `getString` and `getInt`, to retrieve the data from the result sets.

Table 64: Return values for DatabaseMetaData methods

Method	Description	Returns
`allProceduresAreCallable()`	Can all the procedures returned by `getProcedures` be called by the current user?	`False`
`allTablesAreSelectable()`	Can all the tables returned by `getTable` be `SELECT`ed by the current user?	`False`
`dataDefinitionCausesTransactionCommit()`	Does a data definition statement within a transaction force the transaction to commit?	`True`
`dataDefinitionIgnoredInTransactions()`	Is a data definition statement within a transaction ignored?	`False`
`doesMaxRowSizeIncludeBlobs()`	Did `getMaxRowSize( )` include `LONGVARCHAR` and `LONGVARBINARY` `BLOB`s?	`False`
`getBestRowIdentifier (String, String, String, int, boolean)`	Gets a description of a table's optimal set of columns that uniquely identifies a row.	(Result set)
`getCatalogs()`	Gets the catalog names available in this database.	(Result set)
`getCatalogSeparator()`	What is the separator between catalog and table names?	NoneNo catalogs
`getCatalogTerm()`	What is the database vendor's preferred term for `catalog`?	NoneNo catalogs
`getColumnPrivileges (String, String, String, String)`	Gets a description of the access rights for a table's columns.	(Result set)
`getColumns(String, String, String, String)`	Gets a description of table columns available in a catalog.	(Result set)

Method	Description	Returns
`getCrossReference(String, String, String, String, String, String)`	Gets a description of the foreign key columns in the foreign key table that reference the primary key columns of the primary key table (describes how one table imports another's key). This should normally return a single foreign key/primary key pair (most tables only import a foreign key from a table once). They are ordered by `FKTABLE_CAT`, `FKTABLE_SCHEM`, `FKTABLE_NAME`, and `KEY_SEQ`.	(Result set)
`getDatabaseProductName()`	What is the name of this database product?	`OPENEDGE`
`getDatabaseProductVersion()`	What is the version of this database product?	`10.0A1B`
`getDefaultTransactionIsolation()`	What is the database's default transaction isolation level? The values are defined in `java.sql.Connection`.	`TRANSACTION_READ_COMMITTED`
`getDriverMajorVersion()`	What is the version of this JDBC driver?	`1`
`getDriverMinorVersion()`	What is the minor version of this JDBC driver?	`1000`
`getDriverName()`	What is the name of this JDBC driver?	OpenEdge
`getDriverVersion()`	What is the version of this JDBC driver?	`4.0.00 5805 (040318.014802)`
`getExportedKeys(String, String, String)`	Gets a description of the foreign key columns that reference a table's primary key columns (the foreign keys exported by a table).	(Result set)
`getExtraNameCharacters()`	Gets all the extra characters that can be used in unquoted identifier names (those beyond `a-z`, `A-Z`, `0-9` and _).	"_", "%"
`getIdentifierQuoteString ()`	What is the string used to quote SQL identifiers? This returns a space " " if identifier quoting is not supported.	" "
`getImportedKeys(String, String, String)`	Gets a description of the primary key columns that reference a table's foreign key columns (the primary keys imported by a table).	(Result set)

Method	Description	Returns
`getIndexInfo (String, String, String, boolean, boolean)`	Gets a description of a table's indices and statistics.	(Result set)
`getMaxBinaryLiteralLength()`	How many hex characters can you have in an inline binary literal?	`31983`
`getMaxCatalogNameLength()`	What is the maximum length of a catalog name?	NoneNo catalogs
`getMaxCharLiteralLength()`	What is the maximum length for a character literal?	`31983`
`getMaxColumnNameLength()`	What is the limit on column name length?	`32`
`getMaxColumnsInGroupBy()`	What is the maximum number of columns in a `GROUP BY` clause?	`499`
`getMaxColumnsInIndex()`	What is the maximum number of columns allowed in an index?	`16`
`getMaxColumnsInOrderBy()`	What is the maximum number of columns in an `ORDER BY` clause?	`0`
`getMaxColumnsInSelect()`	What is the maximum number of columns in a `SELECT` list?	`500`
`getMaxColumnsInTable()`	What is the maximum number of columns in a table?	`500`
`getMaxConnections()`	How many active connections can we have at a time to this database?	`0`
`getMaxCursorNameLength()`	What is the maximum cursor name length?	`18`
`getMaxIndexLength()`	What is the maximum length of an index (in bytes)?	`113`
`getMaxProcedureNameLength()`	What is the maximum length of a procedure name?	`32`
`getMaxRowSize()`	What is the maximum length of a single row?	`31,995` bytes
`getMaxSchemaNameLength()`	What is the maximum length allowed for a schema name?	`32`
`getMaxStatementLength()`	What is the maximum length of an SQL statement?	`131,000`

Method	Description	Returns
`getMaxStatements()`	How many active statements can we have open at one time to this database?	`100`
`getMaxTableNameLength()`	What is the maximum length of a table name?	`32`
`getMaxTablesInSelect()`	What is the maximum number of tables in a `SELECT`?	`250`
`getMaxUserNameLength()`	What is the maximum length of a user name?	`32`
`getNumericFunctions()`	Gets a comma-separated list of math functions.	`ABS, ACOS, ASIN, ATAN, ATAN2, CEILING, COS, DEGREES, EXP, FLOOR, LOG10, MOD, PI, POWER, RADIANS, RAND, ROUND, SIGN,SIN, SQRT, TAN`
`getPrimaryKeys(String, String, String)`	Gets a description of a table's primary key columns.	(Result set)
`getProcedureColumns (String, String, String, String)`	Get a description of a catalog's stored procedure parameters and result columns.	(Result set)
`getProcedures(String, String, String)`	Gets a description of stored procedures available in a catalog.	(Result set)
`getProcedureTerm()`	What is the database vendor's preferred term for `procedure`?	`procedure`
`getSchemas()`	Gets the schema names available in this database.	(Result set)
`getSchemaTerm()`	What is the database vendor's preferred term for `schema`?	`Owner`
`getSearchStringEscape()`	This is the string that can be used to escape '_' or '%' in the string pattern style catalog search parameters.	`\`
`getSQLKeywords()`	Gets a comma-separated list of all a database's SQL keywords that are NOT also SQL keywords.	See OpenEdge SQL reserved words on page 269 for a complete list of reserved words.
`getStringFunctions()`	Gets a comma-separated list of string functions.	`ASCII, CHAR, CONCAT, DIFFERENCE, INSERT, LCASE, LEFT, LENGTH, LOCATE, LOCATE-2, LTRIM, REPEAT, REPLACE, RIGHT, RTRIM, SPACE, SUBSTRING, UCASE`

Method	Description	Returns
`getSystemFunctions()`	Gets a comma-separated list of system functions.	`USERNAME, IFNULL, DBNAME`
`getTablePrivileges(String, String, String)`	Gets a description of the access rights for each table available in a catalog.	(Result set)
`getTables(String, String, String, String [])`	Gets a description of tables available in a catalog.	(Result set)
`getTableTypes()`	Gets the table types available in this database.	(Result set)
`getTimeDateFunctions()`	Gets a comma-separated list of time and date functions.	`CURDATE, CURTIME, DAYNAME, DAYOFMONTH, DAYOFWEEK, DAYOFYEAR, MONTH, QUARTER, WEEK, YEAR, HOUR, MINUTE, SECOND, MONTHNAME, NOW, TIMESTAMPADD, TIMESTAMPDIFF`
`getTypeInfo()`	Gets a description of all the standard SQL types supported by this database.	(Result set)
`getURL()`	What is the URL for this database?	(The URL)
`getUserName()`	What is our user name as known to the database?	(User name)
`getVersionColumns(String, String, String)`	Gets a description of a table's columns that are automatically updated when any value in a row is updated.	(Result set)
`isCatalogAtStart()`	Does a catalog appear at the start of a qualified table name? Otherwise it appears at the end.	`False`
`isReadOnly()`	Is the database in read-only mode?	`False`
`nullPlusNonNullIsNull()`	Are concatenations between `NULL` and non-`NULL` values `NULL`? A JDBC-compliant driver always returns true.	`True`
`nullsAreSortedAtEnd()`	Are `NULL` values sorted at the end regardless of sort order?	`False`
nullsAreSortedAtStart()	Are `NULL` values sorted at the start regardless of sort order?	`False`
`nullsAreSortedHigh()`	Are `NULL` values sorted high?	`True`
`nullsAreSortedLow()`	Are `NULL` values sorted low?	`False`

Method	Description	Returns
storesLowerCaseIdentifiers()	Does the database treat mixed-case, unquoted SQL identifiers as case insensitive and store them in lowercase?	False
storesLowerCaseQuotedIdentifiers()	Does the database treat mixed-case, quoted SQL identifiers as case insensitive and store them in lowercase?	False
storesMixedCaseIdentifiers()	Does the database treat mixed-case, unquoted SQL identifiers as case insensitive and store them in mixed case?	False
storesMixedCaseQuotedIdentifiers()	Does the database treat mixed-case, quoted SQL identifiers as case insensitive and store them in mixed case?	True
storesUpperCaseIdentifiers()	Does the database treat mixed-case, unquoted SQL identifiers as case insensitive and store them in uppercase?	True
storesUpperCaseQuotedIdentifiers()	Does the database treat mixed-case, quoted SQL identifiers as case insensitive and store them in uppercase?	False
supportsAlterTableWithAddColumn()	Is ALTER TABLE with add column supported?	False
supportsAlterTableWithDropColumn()	Is ALTER TABLE with drop column supported?	False
supportsANSI92EntryLevelSQL()	Is the ANSI92 entry level SQL grammar supported? All JDBC-compliant drivers must return true.	True
supportsANSI92FullSQL()	Is the ANSI92 full SQL grammar supported?	False
supportsANSI92IntermediateSQL()	Is the ANSI92 intermediate SQL grammar supported?	False
supportsCatalogsInDataManipulation()	Can a catalog name be used in a data manipulation statement?	False
supportsCatalogsInIndexDefinitions()	Can a catalog name be used in an index definition statement?	False

Method	Description	Returns
supportsCatalogsInPrivilegeDefinitions()	Can a catalog name be used in a privilege definition statement?	False
supportsCatalogsInProcedureCalls()	Can a catalog name be used in a procedure call statement?	False
supportsCatalogsInTableDefinitions()	Can a catalog name be used in a table definition statement?	False
supportsColumnAliasing()	Is column aliasing supported? If so, the SQL AS clause can be used to provide names for computed columns or to provide alias names for columns as required.	True
supportsConvert()	Is the CONVERT function between SQL types supported?	True
supportsConvert(int, int)	Is CONVERT between the given SQL types supported?	True
supportsCoreSQLGrammar()	Is the ODBC Core SQL grammar supported?	True
supportsCorrelatedSubqueries()	Are correlated subqueries supported? A JDBC-compliant driver always returns true.	True
supportsDataDefinitionAndDataManipulationTransactions ()	Are both data definition and data manipulation statements within a transaction supported?	True
supportsDataManipulationTransactionsOnly()	Are only data manipulation statements within a transaction supported?	False
supportsDifferentTableCorrelationNames()	If table correlation names are supported, are they restricted to be different from the names of the tables?	True
supportsExpressionsInOrderBy()	Are expressions in ORDER BY lists supported?	True
supportsExtendedSQLGrammar()	Is the ODBC Extended SQL grammar supported?	True
supportsFullOuterJoins()	Are full nested outer joins supported?	False
supportsGroupBy()	Is some form of GROUP BY clause supported?	True

Method	Description	Returns
supportsGroupByBeyondSelect()	Can a GROUP BY clause add columns not in the SELECT provided it specifies all the columns in the SELECT?	True
supportsGroupByUnrelated()	Can a GROUP BY clause use columns not in the SELECT?	False
supportsIntegrityEnhancementFacility()	Is the SQL Integrity Enhancement Facility supported?	True
supportsLikeEscapeClause()	Is the escape character in LIKE clauses supported? A JDBC-compliant driver always returns true.	True
supportsLimitedOuterJoins()	Is there limited support for outer joins? (This will be true if supportFullOuterJoins is true.)	False
supportsMinimumSQLGrammar()	Is the ODBC Minimum SQL grammar supported? All JDBC-compliant drivers must return true.	True
supportsMixedCaseIdentifiers()	Does the database treat mixed-case, unquoted SQL identifiers as case sensitive and as a result store them in mixed case? A JDBC-compliant driver will always return false.	False
supportsMixedCaseQuotedIdentifiers()	Does the database treat mixed-case, quoted SQL identifiers as case sensitive and as a result store them in mixed case? A JDBC-compliant driver will always return true.	True
supportsMultipleResultSets()	Are multiple ResultSets from a single execute supported?	False
supportsMultipleTransactions()	Can multiple transactions be open at once (on different connections)?	True
supportsNonNullableColumns()	Can columns be defined as non-nullable? A JDBC-compliant driver always returns true.	True
supportsOpenCursorsAcrossCommit()	Can cursors remain open across commits?	True
supportsOpenCursorsAcrossRollback()	Can cursors remain open across rollbacks?	True
supportsOpenStatementsAcrossCommit()	Can statements remain open across commits?	True

Method	Description	Returns
supportsOpenStatementsAcrossRollback()	Can statements remain open across rollbacks?	True
supportsOrderByUnrelated()	Can an ORDER BY clause use columns not in the SELECT?	False
supportsOuterJoins()	Is some form of outer join supported?	True
supportsPositionedDelete()	Is positioned DELETE supported?	True
supportsPositionedUpdate()	Is positioned UPDATE supported?	True
supportsResultSetType	Retrieves whether this database supports the given result set type. The type may be one of the following: • TYPE_SCROLL_SENSITIVE • TYPE_SCROLL_INSENSITIVE • TYPE_FORWARD_ONLY	True
supportsSchemasInDataManipulation()	Can a schema name be used in a data manipulation statement?	True
supportsSchemasInIndexDefinitions()	Can a schema name be used in an index definition statement?	True
supportsSchemasInPrivilegeDefinitions()	Can a schema name be used in a privilege definition statement?	True
supportsSchemasInProcedureCalls()	Can a schema name be used in a procedure call statement?	True
supportsSchemasInTableDefinitions()	Can a schema name be used in a table definition statement?	True
supportsSelectForUpdate()	Is SELECT for UPDATE supported?	True
supportsStoredProcedures()	Are stored procedure calls using the stored procedure escape syntax supported?	True
supportsSubqueriesInComparisons()	Are subqueries in comparison expressions supported? A JDBC-compliant driver always returns true.	True
supportsSubqueriesInExists()	Are subqueries in EXISTS expressions supported? A JDBC-compliant driver always returns true.	True

Method	Description	Returns
supportsSubqueriesInIns()	Are subqueries in IN statements supported? A JDBC-compliant driver always returns true.	True
supportsSubqueriesInQuantifieds()	Are subqueries in quantified expressions supported? A JDBC-compliant driver always returns true.	True
supportsTableCorrelationNames()	Are table correlation names supported? A JDBC-compliant driver always returns true.	True
supportsTransactionIsolationLevel(int)	Does the database support the given transaction isolation level?	True (for all four transaction levels)
supportsTransactions ()	Are transactions supported? If not, commit is a no-op and the isolation level is TRANSACTION_NONE.	True
supportsUnion()	Is SQL UNION supported?	True
supportsUnionAll()	Is SQL UNION ALL supported?	True
usesLocalFilePerTable()	Does the database use a file for each table?	False
usesLocalFiles()	Does the database store tables in a local file?	False

Example

The following example is a code segment that illustrates calling methods of DatabaseMetadata:

```
Connection con = DriverManager.getConnection ( url, prop);
                                .
                                .
                                .
            // Get the DatabaseMetaData object and display
            // some information about the connection

            DatabaseMetaData dma = con.getMetaData ();

            o.println("\nConnected to " + dma.getURL());
            o.println("Driver       " +
            dma.getDriverName());
            o.println("Version      " +
             dma.getDriverVersion());
```

III

ODBC Reference

For details, see the following topics:

- OpenEdge SQL and ODBC data types
- SQLGetInfo
- ODBC scalar functions

14

OpenEdge SQL and ODBC data types

The following table shows how the OpenEdge SQL data types are mapped to the standard ODBC data types:

Table 65: OpenEdge SQL and ODBC data types

Progress data type	ODBC data type
BINARY	SQL_BINARY
BIT	SQL_BIT
CHAR	SQL_CHAR
DATE	SQL_TYPE_DATE
DECIMAL	SQL_DECIMAL
DOUBLE PRECISION	SQL_DOUBLE
FLOAT	SQL_FLOAT
INTEGER	SQL_INTEGER
REAL	SQL_FLOAT
SMALLINT	SQL_SMALLINT
TIME	SQL_TYPE_TIME

Progress data type	ODBC data type
TIMESTAMP	SQL_TYPE_TIMESTAMP
TINYINT	SQL_TINYINT
VARBINARY	SQL_VARBINARY
LVARBINARY	SQL_LONGVARBINARY
VARCHAR	SQL_VARCHAR

15

SQLGetInfo

This topic details the information the ODBC Driver returns to `SQLGetInfo`.

The following table describes return values the ODBC driver returns to `SQLGetInfo`.

Table 66: Information the ODBC driver returns to SQLGetInfo

Description	fInfoType argument	Returns
Guaranteed execute privileges on all procedures returned by SQLProcedures	`SQL_ACCESSIBLE_PROCEDURES`	`N`
Guaranteed read access to all table names returned by SQLTables	`SQL_ACCESSIBLE_TABLES`	`N`
Maximum number of active connections	`SQL_ACTIVE_CONNECTIONS`	`0`
Maximum number of active statements supported for an active connection	`SQL_ACTIVE_STATEMENTS`	`100`
Maximum number of active environments	`SQL_ACTIVE_ENVIRONMENTS`	`0`
Support for ALTER DOMAIN statement	`SQL_ALTER_DOMAIN`	`0x00000000`
Support for ALTER TABLE clauses	`SQL_ALTER_TABLE`	`0x00000000`
SQL Conformance	`SQL_SQL_CONFORMANCE`	`SQL_SC_SQL92_ENTRY`

Description	fInfoType argument	Returns
Support for datetime literals	SQL_DATETIME_LITERALS	0x00000000
Level of asynchronous mode support	SQL_ASYNC_MODE	SQL_AM_NONE
Behavior with respect to the availability of row counts in batches	SQL_BATCH_ROW_COUNT	0x00000000
Support for batches	SQL_BATCH_SUPPORT	0x00000000
Support for bookmarks	SQL_BOOKMARK_PERSISTENCE	SQL_BP_UPDATE SQL_BP_SCROLL
Position of qualifier in a qualified table name	SQL_CATALOG_LOCATION	SQL_CL_START
Support for catalog names	SQL_CATALOG_NAME	Y
Character used to separate table, column qualifiers	SQL_CATALOG_NAME_SEPARATOR	" ."
Term for object that qualifies table names	SQL_CATALOG_TERM	"database"
Statements that support qualifiers	SQL_CATALOG_USAGE	SQL_CU_DML_STATEMENTS SQL_CU_PROCEDURE_INVOCATION
Default collation sequence name for the default character set	SQL_COLLATION_SEQ	" "
Support for column aliases	SQL_COLUMN_ALIAS	Y
Result of concatenation of NULL character column with non-NULL column	SQL_CONCAT_NULL_BEHAVIOR	SQL_CB_NULL = 0
Conversion from BIGINT	SQL_CONVERT_BIGINT	SQL_CVT_CHAR SQL_CVT_BIGINT SQL_CVT_TINYINT SQL_CVT_SMALLINT SQL_CVT_INTEGER SQL_CVT_FLOAT SQL_CVT_DOUBLE
Conversion from BINARY	SQL_CONVERT_BINARY	0x00000000
Conversion from BIT	SQL_CONVERT_BIT	0x00000000

Description	fInfoType argument	Returns
Conversion from CHAR	SQL_CONVERT_CHAR	SQL_CVT_CHAR SQL_CVT_NUMERIC SQL_CVT_DECIMAL SQL_CVT_INTEGER SQL_CVT_SMALLINT SQL_CVT_FLOAT SQL_CVT_REAL SQL_CVT_DOUBLE SQL_CVT_VARCHAR SQL_CVT_TINYINT SQL_CVT_BIGINT SQL_CVT_DATE SQL_CVT_TIMESTAMP
Conversion from DATE	SQL_CONVERT_DATE	SQL_CVT_CHAR SQL_CVT_VARCHAR SQL_CVT_DATE SQL_CVT_TIMESTAMP
Conversion from DECIMAL	SQL_CONVERT_DECIMAL	SQL_CVT_CHAR SQL_CVT_NUMERIC SQL_CVT_DECIMAL SQL_CVT_INTEGER SQL_CVT_SMALLINT SQL_CVT_FLOAT SQL_CVT_REAL SQL_CVT_DOUBLE SQL_CVT_VARCHAR SQL_CVT_TINYINT SQL_CVT_BIGINT

Description	fInfoType argument	Returns
Conversion from DOUBLE	SQL_CONVERT_DOUBLE	SQL_CVT_CHAR SQL_CVT_NUMERIC SQL_CVT_DECIMAL SQL_CVT_INTEGER SQL_CVT_SMALLINT SQL_CVT_FLOAT SQL_CVT_REAL SQL_CVT_DOUBLE SQL_CVT_VARCHAR SQL_CVT_TINYINT SQL_CVT_BIGINT
Conversion from FLOAT	SQL_CONVERT_FLOAT	SQL_CVT_CHAR SQL_CVT_NUMERIC SQL_CVT_DECIMAL SQL_CVT_INTEGER SQL_CVT_SMALLINT SQL_CVT_FLOAT SQL_CVT_REAL SQL_CVT_DOUBLE SQL_CVT_VARCHAR SQL_CVT_TINYINT SQL_CVT_BIGINT
Support for conversion functions	SQL_CONVERT_FUNCTIONS	SQL_FN_CVT_CONVERT
Conversion from INTEGER	SQL_CONVERT_INTEGER	SQL_CVT_CHAR SQL_CVT_NUMERIC SQL_CVT_DECIMAL SQL_CVT_INTEGER SQL_CVT_SMALLINT SQL_CVT_FLOAT SQL_CVT_REAL SQL_CVT_DOUBLE SQL_CVT_VARCHAR SQL_CVT_TINYINT SQL_CVT_BIGINT

Description	fInfoType argument	Returns
Conversion from INTERVAL_DAY_TIME	SQL_CONVERT_INTERVAL_DAY_TIME	0x00000000
Conversion from INTERVAL_YEAR_MONTH	SQL_CONVERT_INTERVAL_YEAR_MONTH	0x00000000
Conversion from INTERVAL_DAY_TIME	SQL_CONVERT_INTERVAL_DAY_TIME	0x00000000
Conversion from LONGVARBINARY.	SQL_CONVERT_LONGVARBINARY	0x00000000
Conversion from LONGVARCHAR	SQL_CONVERT_LONGVARCHAR	0x00000000
Conversion from NUMERIC	SQL_CONVERT_NUMERIC	SQL_CVT_CHAR SQL_CVT_NUMERIC SQL_CVT_DECIMAL SQL_CVT_INTEGER SQL_CVT_SMALLINT SQL_CVT_FLOAT SQL_CVT_REAL SQL_CVT_DOUBLE SQL_CVT_VARCHAR SQL_CVT_TINYINT SQL_CVT_BIGINT
Conversion from REAL	SQL_CONVERT_REAL	SQL_CVT_CHAR SQL_CVT_NUMERIC SQL_CVT_DECIMAL SQL_CVT_INTEGER SQL_CVT_SMALLINT SQL_CVT_FLOAT SQL_CVT_REAL SQL_CVT_DOUBLE SQL_CVT_VARCHAR SQL_CVT_TINYINT SQL_CVT_BIGINT

Description	fInfoType argument	Returns
Conversion from SMALLINT	SQL_CONVERT_SMALLINT	SQL_CVT_CHAR SQL_CVT_NUMERIC SQL_CVT_DECIMAL SQL_CVT_INTEGER SQL_CVT_SMALLINT SQL_CVT_FLOAT SQL_CVT_REAL SQL_CVT_DOUBLE SQL_CVT_VARCHAR SQL_CVT_TINYINT SQL_CVT_BIGINT
Conversion from TIME	SQL_CONVERT_TIME	SQL_CVT_CHAR SQL_CVT_TIME SQL_CVT_TIMESTAMP
Conversion from TIMESTAMP	SQL_CONVERT_TIMESTAMP	SQL_CVT_CHAR SQL_CVT_VARCHAR SQL_CVT_DATE SQL_CVT_TIME SQL_CVT_TIMESTAMP
Conversion from TINYINT	SQL_CONVERT_TINYINT	SQL_CVT_CHAR SQL_CVT_NUMERIC SQL_CVT_DECIMAL SQL_CVT_INTEGER SQL_CVT_SMALLINT SQL_CVT_FLOAT SQL_CVT_REAL SQL_CVT_DOUBLE SQL_CVT_VARCHAR SQL_CVT_TINYINT SQL_CVT_BIGINT
Conversion from VARBINARY	SQL_CONVERT_VARBINARY	0x00000000

Description	fInfoType argument	Returns
Conversion from VARCHAR	SQL_CONVERT_VARCHAR	SQL_CVT_CHAR SQL_CVT_NUMERIC SQL_CVT_DECIMAL SQL_CVT_INTEGER SQL_CVT_SMALLINT SQL_CVT_FLOAT SQL_CVT_REAL SQL_CVT_DOUBLE SQL_CVT_VARCHAR SQL_CVT_TINYINT SQL_CVT_BIGINT SQL_CVT_DATE SQL_CVT_TIME SQL_CVT_TIMESTAMP
Conversion from WCHAR	SQL_CONVERT_WCHAR	0x00000000
Conversion from WLONGVARCHAR	SQL_CONVERT_WLONGVARCHAR	0x00000000
Conversion from WVARCHAR	SQL_CONVERT_WVARCHAR	0x00000000
Support for table correlation names	SQL_CORRELATION_NAME	SQL_CN_DIFFERENT
Support for CREATE ASSERTIONstatement	SQL_CREATE_ASSERTION	0x00000000
Support for CREATE CHARACTER SET statement.	SQL_CREATE_CHARACTER SET	0x00000000
Support for CREATE COLLATION statement	SQL_CREATE_COLLATION	0x00000000
Support for CREATE DOMAIN statement	SQL_CREATE_DOMAIN	0x00000000
Support for CREATE SCHEMA statement	SQL_CREATE_SCHEMA	0x00000000
Support for CREATE TABLE statement	SQL_CREATE_TABLE	SQL_CT_CREATE_TABLE SQL_CT_COLUMN_CONSTRAINT SQL_CT_TABLE_CONSTRAINT
Support for CREATE TRANSLATIONstatement	SQL_CREATE_TRANSLATION	0x00000000

Description	fInfoType argument	Returns
Support for CREATE VIEW statement	SQL_CREATE_VIEW	SQL_CV_CREATE_VIEW SQL_CV_CHECK_OPTION
Effect of COMMIT operation on cursors and prepared statements	SQL_CURSOR_COMMIT_BEHAVIOR	SQL_CB_PRESERVE
Effect of ROLLBACK operation on cursors and prepared statements	SQL_CURSOR_ROLLBACK_BEHAVIOR	SQL_CB_PRESERVE
Support for cursor sensitivity	SQL_CURSOR_SENSITIVITY	SQL_INSENSITIVE
Name of the data source as specified to the ODBC Administrator	SQL_DATA_SOURCE_NAME	(String containing the name)
Access limited to read-only	SQL_DATA_SOURCE_READ_ONLY	N (Read-write access)
Name of the Progress SQL-92 ODBC data source on the server system	SQL_DATABASE_NAME	(String containing the name)
Name of the database product supporting the data source	SQL_DBMS_NAME	OPENEDGE
Version of the database product	SQL_DBMS_VER	10.1B
Default transaction isolation level	SQL_DEFAULT_TXN_ISOLATION	SQL_TXN_READ_COMMITTED
Support for describing parameters via DESCRIBE INPUT statement	SQL_DESCRIBE_PARAMETER	Y (Supports)
Version of the driver manager	SQL_DM_VER	03.52.1117.0000
Connection handle determined by the argument InfoType	SQL_DRIVER_HDBC	0x017E4538
Driver's descriptor handle determined by the Driver Manager's descriptor handle	SQL_DRIVER_HDESC	0x017E68A8
Environment handle determined by the argument InfoType	SQL_DRIVER_HENV	0x017E4090
Handle library from the load library returned to the Driver Manager when it loaded the driver DLL	SQL_DRIVER_HLIB	0x28660000
Driver's statement handle determined by the Driver Manager's statement handle	SQL_DRIVER_HSTMT	0x01828050

Description	fInfoType argument	Returns
Name of the dynamic link library file for the ODBC Driver	SQL_DRIVER_NAME	Windows pgoe1022.DLL AIX, SOLARIS, LINUX pgoe1022.SO HPIIX pgoe1022.SL
Supported ODBC version	SQL_DRIVER_ODBC_VER	03.52
Current version of the ODBC Driver	SQL_DRIVER_VER	05.20.0039 (b0034, u0022)
Support for DROP ASSERTION statement	SQL_DROP_ASSERTION	0x00000000
Support for DROP CHARACTER SET statement	SQL_DROP_CHARACTER_SET	0x00000000
Support for DROP COLLATION statement	SQL_DROP_COLLATION	0x00000000
Support for *D*ROP DOMAIN statement	SQL_DROP_DOMAIN	0x00000000
Support for DROP SCHEMA statement	SQL_DROP_SCHEMA	0x00000000
Support for *DROP TABLE* statement	SQL_DROP_TABLE	SQL_DT_DROP_TABLE
Support for *DROP TRANSLATION* statement	SQL_DROP_TRANSLATION	0x00000000
Support for DROP VIEW statement	SQL_DROP_VIEW	SQL_DV_DROP_VIEW
Supported attributes of a dynamic cursor: subset 1	SQL_DYNAMIC_CURSOR_ ATTRIBUTES1	0x00000000
Supported attributes of a dynamic cursor: subset 2	SQL_DYNAMIC_CURSOR_ ATTRIBUTES2	0x00000000
Support for expressions in ORDER BY clause	SQL_EXPRESSIONS_IN_ORDERBY	Y

Description	fInfoType argument	Returns
Supported fetch direction option.	SQL_FETCH_DIRECTION	SQL_FD_FETCH_NEXT SQL_FD_FETCH_FIRST SQL_FD_FETCH_LAST SQL_FD_FETCH_PRIOR SQL_FD_FETCH_ABSOLUTE SQL_FD_FETCH_RELATIVE SQL_FD_FETCH_BOOKMARK
Single-tier driver behavior	SQL_FILE_USAGE	SQL_FILE_NOT_SUPPORTED
Supported attributes of a forward-only cursor: subset 1	SQL_FORWARD_ONLY_ CURSOR_ATTRIBUTES1	SQL_CA1_NEXT SQL_CA1_BULK_ADD
Supported attributes of a forward-only cursor: subset 2	SQL_FORWARD_ONLY_ CURSOR_ATTRIBUTES2	SQL_CA2_MAX_ROWS_SELECT SQL_CA2_MAX_ROWS_CATALOG
Supported extensions to SQLGetData	SQL_GETDATA_EXTENSIONS	SQL_GD_ANY_COLUMN SQL_GD_ANY_ORDER SQL_GD_BLOCK SQL_GD_BOUND
Relationship between GROUP BY clause and columns in the select list	SQL_GROUP_BY	SQL_GB_GROUP_BY_CONTAINS_SELECT
Case sensitivity of user-supplied names	SQL_IDENTIFIER_CASE	SQL_IC_UPPER
Character used to enclose delimited identifiers	SQL_IDENTIFIER_QUOTE_CHAR	" (Double quotation mark)
Keywords supported in the CREATE INDEX statement	SQL_INDEX_KEYWORDS	SQL_IK_ASC | SQL_IK_DESC
Supported views in INFORMATION_SCHEMA	SQL_INFO_SCHEMA_VIEWS	0x00000000
Support for Integrity Enhancement Facility	SQL_INTEGRITY	Y

Description	fInfoType argument	Returns
Supported attributes of a keyset cursor: subset 1	SQL_KEYSET_CURSOR_ATTRIBUTES1	SQL_CA1_NEXT SQL_CA1_ABSOLUTE SQL_CA1_RELATIVE SQL_CA1_BOOKMARK SQL_CA1_LOCK_NO_CHANGE SQL_CA1_POS_POSITION SQL_CA1_POS_UPDATE SQL_CA1_POS_DELETE SQL_CA1_POS_REFRESH SQL_CA1_POSITIONED_UPDATE SQL_CA1_POSITIONED_DELETE SQL_CA1_SELECT_FOR_UPDATE SQL_CA1_BULK_ADD
Supported attributes of a keyset cursor: subset 2	SQL_KEYSET_CURSOR_ATTRIBUTES2	SQL_CA2_READ_ONLY_CONCURRENCY SQL_CA2_OPT_VALUES_CONCURRENCY SQL_CA2_SENSITIVITY_DELETIONS SQL_CA2_SENSITIVITY_UPDATES SQL_CA2_MAX_ROWS_SELECT SQL_CA2_CRC_EXACT SQL_CA2_SIMULATE_TRY_UNIQUE
Data source specific keywords	SQL_KEYWORDS	See OpenEdge SQL reserved words on page 269 for a list of SQL Keywords.
Support for escape clause in LIKE predicates	SQL_LIKE_ESCAPE_CLAUSE	Y
Support for lock types	SQL_LOCK_TYPES	SQL_LCK_NO_CHANGE
Maximum number of active concurrent statements in asynchronous mode	SQL_MAX_ASYNC_CONCURRENT_STATEMENTS	0
Maximum length in hexadecimal characters of binary literals	SQL_MAX_BINARY_LITERAL_LEN	31,983
Maximum length of a table or column qualifier	SQL_MAX_CATALOG_NAME_LEN	32
Maximum length in characters of character string literals	SQL_MAX_CHAR_LITERAL_LEN	31,983

Description	fInfoType argument	Returns
Maximum length of a column name	SQL_MAX_COLUMN_NAME_LEN	32
Maximum number of columns allowed in GROUP BY clause	SQL_MAX_COLUMNS_IN_GROUP_BY	5000
Maximum number of columns allowed in an index	SQL_MAX_COLUMNS_IN_INDEX	16
Maximum number of columns allowed in ORDER BY clause	SQL_MAX_COLUMNS_IN_ORDER_BY	0
Maximum number of columns allowed in a SELECT list	SQL_MAX_COLUMNS_IN_SELECT	5000
Maximum number of columns allowed in a table	SQL_MAX_COLUMNS_IN_TABLE	5000
Maximum number of active SQL statements	SQL_MAX_CONCURRENT_ACTIVITIES	100
Maximum length of a cursor name	SQL_MAX_CURSOR_NAME_LEN	18
Maximum number of active connections	SQL_MAX_DRIVER_CONNECTIONS	0
Maximum length of user-defined names	SQL_MAX_IDENTIFIER_LEN	32
Maximum number of bytes allowed in the combined fields of an index	SQL_MAX_INDEX_SIZE	115
Maximum length of a procedure name	SQL_MAX_PROCEDURE_NAME_LEN	32
Maximum length in bytes of a table row	SQL_MAX_ROW_SIZE	31995
Whether maximum row size includes *LONGVARCHAR* and *LONGVARBINARY*	SQL_MAX_ROW_SIZE_INCLUDES_LONG	N
Maximum length of an owner name	SQL_MAX_SCHEMA_NAME_LEN	32
Maximum number of characters in an SQL statement	SQL_MAX_STATEMENT_LEN	131,000
Maximum length of a table name	SQL_MAX_TABLE_NAME_LEN	32
Maximum number of tables allowed in FROM clause	SQL_MAX_TABLES_IN_SELECT	250
Maximum length of a user name	SQL_MAX_USER_NAME_LEN	32
Maximum length of owner name	SQL_MAX_OWNER_NAME_LEN	32

Description	fInfoType argument	Returns
Maximum length of a qualifier name	SQL_MAX_QUALIFIER_NAME_LEN	32
Support for multiple result sets	SQL_MULT_RESULT_SETS	N
Support for active transactions on multiple connections	SQL_MULTIPLE_ACTIVE_TXN	Y
Whether data source requires length of *LONGVARCHAR* and *LONGVARBINARY* data	SQL_NEED_LONG_DATA_LEN	Y
Support for *NOT NULL* clause in *CREATE TABLE* statement	SQL_NON_NULLABLE_COLUMNS	SQL_NNC_NON_NULL
Where NULL values are sorted in a list	SQL_NULL_COLLATION	SQL_NC_HIGH
Numeric functions supported	SQL_NUMERIC_FUNCTIONS	SQL_FN_NUM_ABS SQL_FN_NUM_ACOS SQL_FN_NUM_ASIN SQL_FN_NUM_ATAN SQL_FN_NUM_ATAN2 SQL_FN_NUM_CEILING SQL_FN_NUM_COS SQL_FN_NUM_EXP SQL_FN_NUM_FLOOR SQL_FN_NUM_MOD SQL_FN_NUM_PI SQL_FN_NUM_POWER SQL_FN_NUM_RADIANS SQL_FN_NUM_RAND SQL_FN_NUM_ROUND SQL_FN_NUM_SIGN SQL_FN_NUM_SIN SQL_FN_NUM_SQRT SQL_FN_NUM_TAN SQL_FN_NUM_DEGREES SQL_FN_NUM_LOG10
Level of ODBC conformance	SQL_ODBC_API_CONFORMANCE	SQL_OAC_LEVEL1
Level of ODBC 3.x interface conformance	SQL_ODBC_INTERFACE_CONFORMANCE	SQL_OIC_CORE

Description	fInfoType argument	Returns
SQL Access Group (SAG) conformance	SQL_ODBC_SAG_CLI_CONFORMANCE	SQL_OSCC_COMPLIANT
Level of SQL conformance	SQL_ODBC_SQL_CONFORMANCE	SQL_OSC_EXTENDED
Referential integrity syntax support	SQL_ODBC_SQL_OPT_IEF	Y
ODBC version supported by driver manager	SQL_ODBC_VER	03.52.0000
Types of outer joins supported	SQL_OJ_CAPABILITIES	SQL_OJ_LEFT SQL_OJ_RIGHT SQL_OJ_NOT_ORDERED SQL_OJ_INNER SQL_OJ_ALL_COMPARISON_OPS
Whether columns in ORDER BY clause must also be in select list	SQL_ORDER_BY_COLUMNS_IN_SELECT	N
Support for outer joins	SQL_OUTER_JOINS	Y
Name for an owner	SQL_OWNER_TERM	owner
Statements in which owner can be used	SQL_OWNER_USAGE	SQL_OU_DML_STATEMENTS SQL_OU_PROCEDURE_INVOCATION SQL_OU_TABLE_DEFINITION SQL_OU_INDEX_DEFINITION SQL_OU_PRIVILEGE_DEFINITION
Characteristics of row counts available in a parameterized execution	SQL_PARAM_ARRAY_ROW_COUNTS	SQL_PARC_NO_BATCH
Characteristics of result sets available in a parameterized execution	SQL_PARAM_ARRAY_SELECTS	SQL_PAS_NO_SELECT
Supported operations in SQLSetPos	SQL_POS_OPERATIONS	SQL_POS_POSITION SQL_POS_REFRESH SQL_POS_UPDATE SQL_POS_DELETE SQL_POS_ADD
Supported positioned SQL statements	SQL_POSITIONED_STATEMENTS	SQL_PS_POSITIONED_DELETE SQL_PS_POSITIONED_UPDATE SQL_PS_SELECT_FOR_UPDATE

Description	fInfoType argument	Returns
Term for procedures	SQL_PROCEDURE_TERM	procedure
SQL procedures support	SQL_PROCEDURES	Y
Support for qualifiers	SQL_QUALIFIER_USAGE	SQL_CU_DML_STATEMENTS SQL_CU_PROCEDURE_INVOCATION
Case sensitivity of quoted user-supplied names	SQL_QUOTED_IDENTIFIER_CASE	SQL_IC_MIXED
Separator character used between qualifier name and element	SQL_QUALIFIER_NAME_SEPARATOR	" . "
Term used for a qualifier	SQL_QUALIFIER_TERM	"database"
Position of the qualifier in a qualified table name	SQL_QUALIFIER_LOCATION	SQL_CL_START
Detect changes to any row in mixed-cursor operations	SQL_ROW_UPDATES	Y
Term for entity that has owner privileges on objects	SQL_SCHEMA_TERM	owner
Statements that support use of owner qualifiers	SQL_SCHEMA_USAGE	SQL_OU_DML_STATEMENTS SQL_OU_PROCEDURE_INVOCATION SQL_OU_TABLE_DEFINITION SQL_OU_INDEX_DEFINITION SQL_OU_PRIVILEGE_DEFINITION
Options supported for scrollable cursors	SQL_SCROLL_OPTIONS	SQL_SO_FORWARD_ONLY SQL_SO_STATIC SQL_SO_KEYSET_DRIVEN
Support for scrollable cursors	SQL_SCROLL_CONCURRENCY	SQL_SCCO_READ_ONLY SQL_SCCO_OPT_VALUES
Character to permit wildcard characters in search strings	SQL_SEARCH_PATTERN_ESCAPE	\ (Backslash)
Name of the system where the ODBC data source resides	SQL_SERVER_NAME	(String containing the name)
Special characters allowed in user-supplied names	SQL_SPECIAL_CHARACTERS	"_", "%"
Datetime scalar functions supported	SQL_SQL92_DATETIME_FUNCTIONS	0x00000000

Description	fInfoType argument	Returns
Behavior of DELETE statement that refers to a foreign key.	SQL_SQL92_FOREIGN_KEY_DELETE_RULE	0x00000000
Behavior of UPDATE statement that refers to a foreign key	SQL_SQL92_FOREIGN_KEY_UPDATE_RULE	0x00000000
GRANT statement clauses supported	SQL_SQL92_GRANT	SQL_SG_DELETE_TABLE SQL_SG_INSERT_TABLE SQL_SG_INSERT_COLUMN SQL_SG_REFERENCES_TABLE SQL_SG_REFERENCES_COLUMN SQL_SG_SELECT_TABLE SQL_SG_UPDATE_TABLE SQL_SG_UPDATE_COLUMN
Numeric scalar functions supported	SQL_SQL92_NUMERIC_ VALUE_FUNCTIONS	SQL_SNVF_CHAR_LENGTH SQL_SNVF_CHARACTER_LENGTH
Predicates supported	SQL_SQL92_PREDICATES	SP_EXISTS SQL_SP_ISNOTNULL SQL_SP_ISNULL SQL_SP_UNIQUE SQL_SP_LIKE SQL_SP_IN SQL_SP_BETWEEN
Relational join operators supported	SQL_SQL92_RELATIONAL_JOIN_OPERATORS	0x00000000
REVOKE statement clauses supported	SQL_SQL92_REVOKE	SQL_SR_GRANT_OPTION_FOR SQL_SR_CASCADE SQL_SR_RESTRICT SQL_SR_DELETE_TABLE SQL_SR_INSERT_TABLE SQL_SR_INSERT_COLUMN SQL_SR_REFERENCES_TABLE SQL_SR_REFERENCES_COLUMN SQL_SR_SELECT_TABLE SQL_SR_UPDATE_TABLE SQL_SR_UPDATE_COLUMN

Description	fInfoType argument	Returns
Row value constructor expressions supported	SQL_SQL92_ROW_VALUE_CONSTRUCTOR	0x00000000
String scalar functions supported	SQL_SQL92_STRING_FUNCTIONS	SQL_SSF_CONVERT SQL_SSF_LOWER SQL_SSF_UPPER SQL_SSF_SUBSTRING SQL_SSF_TRANSLATE SQL_SSF_TRIM_LEADING SQL_SSF_TRIM_TRAILING
Value expressions supported	SQL_SQL92_VALUE_EXPRESSIONS	SQL_SVE_COALESCE SQL_SVE_NULLIF
CLI standards to which the driver conforms	SQL_STANDARD_CLI_CONFORMANCE	SQL_SCC_XOPEN_CLI_VERSION1
Supported attributes of a static cursor: subset 1	SQL_STATIC_CURSOR_ATTRIBUTES1	SQL_CA1_NEXT SQL_CA1_ABSOLUTE SQL_CA1_RELATIVE SQL_CA1_BOOKMARK SQL_CA1_LOCK_NO_CHANGE SQL_CA1_POS_POSITION SQL_CA1_POS_UPDATE SQL_CA1_POS_DELETE SQL_CA1_POS_REFRESH SQL_CA1_POSITIONED_UPDATE SQL_CA1_POSITIONED_DELETE SQL_CA1_SELECT_FOR_UPDATE SQL_CA1_BULK_ADD
Supported attributes of a static cursor: subset 2	SQL_STATIC_CURSOR_ATTRIBUTES2	SQL_CA2_READ_ONLY_CONCURRENCY SQL_CA2_OPT_VALUES_CONCURRENCY SQL_CA2_SENSITIVITY_UPDATES SQL_CA2_MAX_ROWS_SELECT SQL_CA2_CRC_EXACT SQL_CA2_SIMULATE_TRY_UNIQUE

Description	fInfoType argument	Returns
Support for detection of changes made to a static or key-set driven cursor through SQLSetPos	SQL_STATIC_SENSITIVITY	0x00000000
String functions supported	SQL_STRING_FUNCTIONS	SQL_FN_STR_CONCAT S SQL_FN_STR_INSERT SQL_FN_STR_LEFT SQL_FN_STR_LTRIM SQL_FN_STR_LENGTH SQL_FN_STR_LOCATE SQL_FN_STR_LCASE SQL_FN_STR_REPEAT SQL_FN_STR_REPLACE SQL_FN_STR_RIGHT SQL_FN_STR_RTRIM SQL_FN_STR_SUBSTRING SQL_FN_STR_UCASE SQL_FN_STR_ASCII SQL_FN_STR_CHAR SQL_FN_STR_DIFFERENCE SQL_FN_STR_LOCATE_2 SQL_FN_STR_SPACE SQL_FN_STR_CHAR_LENGTH QL_FN_STR_CHARACTER_LENGTH
Predicates that support subqueries	SQL_SUBQUERIES	SQL_SQ_COMPARISON SQL_SQ_EXISTS SQL_SQ_IN SQL_SQ_QUANTIFIED SQL_SQ_CORRELATED_SUBQUERIES
System functions supported	SQL_SYSTEM_FUNCTIONS	SQL_FN_SYS_USERNAME
Term for tables	SQL_TABLE_TERM	"table"
Timestamp intervals supported for TIMESTAMPADD function	SQL_TIMEDATE_ADD_INTERVALS	0x00000000
Timestamp intervals supported for TIMESTAMPDIFF function	SQL_TIMEDATE_DIFF_INTERVALS	0x00000000

Description	fInfoType argument	Returns
Date-time functions supported	SQL_TIMEDATE_FUNCTIONS	SQL_FN_TD_NOW SQL_FN_CURDATE SQL_FN_TD_DAYOFMONTH SQL_FN_TD_DAYOFWEEK SQL_FN_TD_DAYOFYEAR SQL_FN_TD_MONTH SQL_FN_TD_QUARTER SQL_FN_TD_WEEK SQL_FN_TD_YEAR SQL_FN_CURTIME SQL_FN_TD_HOUR SQL_FN_TD_MINUTE SQL_FN_TD_SECOND SQL_FN_TD_TIMESTAMP_ADD SQL_FN_TD_TIMESTAMPDIFF SQL_FN_TD_DAYNAME SQL_FN_TD_MONTHNAME
Support for DML, DDL within transactions	SQL_TXN_CAPABLE	SQL_TC_ALL
Options for setting transaction isolation levels	SQL_TXN_ISOLATION_OPTION	SQL_TXN_READ_UNCOMMITTED SQL_TXN_SERIALIZABLE SQL_TXN_READ_COMMITTED SQL_TXN_REPEATABLE_READ
UNION support	SQL_UNION	SQL_U_UNION SQL_U_UNION_ALL
Name of user connected to the data source	SQL_USER_NAME	(String containing the name)
Year of publication of the X/Open specification with which the driver complies	SQL_XOPEN_CLI_YEAR	1995

16

ODBC scalar functions

This topic lists scalar functions that ODBC supports and are available to use in OpenEdge SQL statements.

For details, see the following topics:

- Scalar functions
- System functions

Scalar functions

The tables in this topic list the scalar functions that ODBC supports.

Syntax

You can use these functions in SQL statements using the following syntax:

```
{fn scalar-function}
```

scalar-function is one of the functions listed in the following tables.

Example

```
SELECT {fn UCASE(NAME)} FROM EMP
```

String functions

The table in this topic lists the string functions that ODBC supports.

The string functions listed can take the following arguments:

- *string_exp* can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type is SQL_CHAR or SQL_VARCHAR.
- *start, length,* and *count* can be the result of another scalar function or a literal numeric value, where the underlying data type is SQL_TINYINT, SQL_SMALLINT, or SQL_INTEGER.

The string functions are one-based; that is, the first character in the string is character 1.

Character string literals must be surrounded in single quotation marks.

Table 67: Scalar string functions

String function	Returns
ASCII(*string_exp*)	ASCII code value of the leftmost character of string_exp as an integer.
BIT_LENGTH(*string_exp*)ODBC 3.0	The length in bits of the string expression.
CHAR(*code*)	The character with the ASCII code value specified bycode. code should be between 0 and 255; otherwise, the return value is data-source dependent.
CHAR_LENGTH(*string_exp*) ODBC 3.0	The length in characters of the string expression, if the string expression is of a character data type; otherwise, the length in bytes of the string expression (the smallest integer not less than the number of bits divided by 8). (This function is the same as the CHARACTER_LENGTH function.)
CHARACTER_LENGTH(*string_exp*)ODBC 3.0	The length in characters of the string expression, if the string expression is of a character data type; otherwise, the length in bytes of the string expression (the smallest integer not less than the number of bits divided by 8). (This function is the same as the CHAR_LENGTH function.)
CONCAT(*string_exp1*,*string_exp*)	The string resulting from concatenating string_exp2 and string_exp1. The string is system dependent.
DIFFERENCE(*string_exp2* and *string_exp1*)	An integer value that indicates the difference between the values returned by the SOUNDEX function for string_exp2 and string_exp1.
INSERT(*string_exp1 , start, length, string_exp2*)	A string where length characters have been deleted from string_exp1 beginning at start and where string_exp2 has been inserted into string_exp, beginning at start.
LCASE(*string_exp*)	Uppercase characters in string_exp converted to lowercase.
LEFT(*string_exp, count*)	The count of characters of string_exp.
LENGTH(*string_exp*)	The number of characters in string_exp.
LOCATE(*string_exp1, string_exp2*[,start,])	

String function	Returns
	The starting position of the first occurrence of `string_exp1` within `string_exp2`. If `start` is not specified the search begins with the first character position in `string_exp2`. If `start` is specified, the search begins with the character position indicated by the value of `start`. The first character position in `string_exp2` is indicated by the value 1. If `string_exp1` is not found, 0 is returned.
`LTRIM(`*`string_exp`*`)`	The characters of `string_exp`, with leading blanks removed.
`OCTET_LENGTH(`*`string_exp`*`)` `ODBC 3.0`	The length in bytes of the string expression. The result is the smallest integer not less than the number of bits divided by 8.
`POSITION(`*`character_exp`* `IN` *`character_exp`*`)`	The position of the first character expression in the second character expression. The result is an exact numeric with an implementation-defined precision and a scale of 0.
`REPEAT(`*`string_exp`*`, count)`	A string composed of `string_exp` repeated *`count`* times.
`REPLACE(`*`string_exp1`*`,` *`string_exp2`*`,` *`string_exp3`*`)`	Replaces all occurrences of `string_exp2` in `string_exp1` with `string_exp3`.
`RIGHT(`*`string_exp`*`, count)`	The rightmost `count` of characters in `string_exp`.
`RTRIM(`*`string_exp`*`)`	The characters of `string_exp` with trailing blanks removed.
`SPACE(count)`	A string consisting of `count` spaces.
`SUBSTRING(`*`string_exp`* `, start, length)`	A string derived from `string_exp` beginning at the character position `start` for `length` characters.
`UCASE(`*`string_exp`*`)`	Lowercase characters in `string_exp` converted to uppercase.

Numeric functions

The table in this topic lists the numeric functions that ODBC supports.

The numeric functions listed can take the following arguments:

- *numeric_exp* can be a column name, a numeric literal, or the result of another scalar function, where the underlying data type is SQL_NUMERIC, SQL_DECIMAL, SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_FLOAT, SQL_REAL, or SQL_DOUBLE.
- *float_exp* can be a column name, a numeric literal, or the result of another scalar function, where the underlying data type is SQL_FLOAT.
- *integer_exp* can be a column name, a numeric literal, or the result of another scalar function, where the underlying data type is SQL_TINYINT, SQL_SMALLINT, or SQL_INTEGER.

Table 68: Scalar numeric functions

Numeric function	Returns
ABS(*numeric_exp*)	Absolute value of *numeric_exp*.
ACOS(*float_exp*)	Arccosine of *float_exp* as an angle in radians.
ASIN(*float_exp*)	Arcsine of *float_exp* as an angle in radians.
ATAN(*float_exp*)	Arctangent of *float_exp* as an angle in radians.
ATAN2(*float_exp1, float_exp2*)	Arctangent of the x and y coordinates, specified by *float_exp1* and *float_exp2* as an angle in radians.
CEILING(*numeric_exp*)	Smallest integer greater than or equal to *numeric_exp*.
COS(*float_exp*)	Cosine of *float_exp* as an angle in radians.
COT(*float_exp*)	Cotangent of *float_exp* as an angle in radians.
DEGREES(*numeric_exp*)	Number of degrees converted from *numeric_exp* radians.
EXP(*float_exp*)	Exponential value of *float_exp*.
FLOOR(*numeric_exp*)	Largest integer less than or equal to *numeric_exp*.
LOG(*float_exp*)	Natural log of *float_exp*.
LOG10(*float_exp*)	Base 10 log of *float_exp*.
MOD(*integer_exp1, integer_exp2*)	Remainder of *integer_exp1* divided by *integer_exp2*.
PI()	Constant value of pi as a floating-point number.

Numeric function	Returns
POWER(*numeric_exp*, *integer_exp*)	Value of *numeric_exp* to the power of *integer_exp*.
RADIANS(*numeric_exp*)	Number of radians converted from *numeric_exp* degrees.
RAND([*integer_exp*])	Random floating-point value using *integer_exp* as the optional seed value.
ROUND(*numeric_exp*, *integer_exp*)	*numeric_exp* rounded to *integer_exp* places right of the decimal (left of the decimal if *integer_exp* is negative).
SIGN(*numeric_exp*)	Indicator of the sign of *numeric_exp*. If *numeric_exp* <0, -1 is returned. If *numeric_exp* = 0, 0 is returned. If *numeric_exp* > 0, 1 is returned.
SIN(*float_exp*)	Sine of *float_exp*, where *float_exp* is an angle in radians.
SQRT(*float_exp*)	Square root of *float_exp*.
TAN(*float_exp*)	Tangent of *float_exp*, where *float_exp* is an angle in radians.
TRUNCATE(*numeric_exp*, *integer_exp*)	*numeric_exp* truncated to *integer_exp* places right of the decimal. (If *integer_exp* is negative, truncation is to the left of the decimal.)

Date and time functions

The table in this topic lists the date and time functions that ODBC supports.

The date and time functions listed can take the following arguments:

- *date-exp* can be a column name, a date or timestamp literal, or the result of another scalar function, where the underlying data type can be represented as SQL_CHAR, SQL_VARCHAR, SQL_DATE, or SQL_TIMESTAMP.
- *time_exp* can be a column name, a timestamp or timestamp literal, or the result of another scalar function, where the underlying data type can be represented as SQL_CHAR, SQL_VARCHAR, SQL_TIME, or SQL_TIMESTAMP.
- *timestamp_exp* can be a column name; a time, date, or timestamp literal; or the result of another scalar function, where the underlying data type can be represented as SQL_CHAR, SQL_VARCHAR, SQL_TIME, SQL_DATE, or SQL_TIMESTAMP.

Table 69: Date and time functions supported by ODBC

Function	Returns
CURRENT DATE() (ODBC 3.6)	Current date.
CURRENT_TIME [(*time-precision*)](ODBC 3.6)	Current local time. The time-precisionargument determines the seconds precision of the returned value.
CURRENT_TIMESTAMP [(*timestamp-precision*)](ODBC 3.6)	Current local date and local time as a timestamp value. The timestamp-precision argument determines the seconds precision of the returned timestamp.
CURDATE()	Current date as a date value.
CURTIME()	Current local time as a time value.
DAYNAME(*date_exp*)	Character string containing a date source-specific name of the day for the day portion of date_exp.
DAYOFMONTH(*date_exp*)	Day of the month in date_exp as an integer value (1-31).
DAYOFWEEK(*date_exp*)	Day of the week in date_exp as an integer value (1-7).
DAYOFYEAR(*date_exp*)	Day of the year in date_exp as an integer value (1-366).
HOUR(*time_exp*)	Hour in time_exp as an integer value (0-23).
MINUTE(*time_exp*)	Minute in time_exp as an integer value (0-59).
MONTH(*date_exp*)	Month in date_exp as an integer value (1-366).
MONTHNAME(*date_exp*)	Character string containing the data source-specific name of the month.
	Current date and time as a timestamp value.

Function	Returns
NOW()	
QUARTER(*date_exp*)	Quarter in date_exp as an integer value (1-4).
SECOND(*time_exp*)	Second in date_exp as an integer value (0-59).
TIMESTAMPADD (*interval*, *integer_exp*, *time_exp*)	Timestamp calculated by adding integer_exp intervals of type interval to time_exp. interval can be: • SQL_TSI_FRAC_SECOND • SQL_TSI_SECOND • SQL_TSI_MINUTE • SQL_TSI_HOUR • SQL_TSI_DAY • SQL_TSI_WEEK • SQL_TSI_MONTH • SQL_TSI_QUARTER • SQL_TSI_YEAR Fractional seconds are expressed in billionths of a second.
TIMESTAMPDIFF (*interval*, *time_exp1*, time_exp2)	Integer number of intervals of type interval by which time_exp2 is greater than time_exp1. interval has the same values as TIMESTAMPADD. Fractional seconds are expressed in billionths of a second.
WEEK(*date_exp*)	Week of the year in date_exp as an integer value (1-53).
YEAR(*date_exp*)	Year in date_exp. The range is data source dependent.

System functions

The following table lists the scalar system functions that ODBC supports.

Table 70: Scalar system functions supported by ODBC

System function	Returns
DATABASE()	Name of the database, corresponding to the connection handle (odbc)
IFNULL(*exp, value*)	value, if exp is null
ROWID(*extension*)	The row identifier of the current row in a table
USER()	Authorization name of the user
tenantname()	Name of the tenant corresponding to the connection handle (odbc).
tenantid()	Unique ID associated with the name of the tenant.
tenantName_tbl ([owner_name. | *catalog_name.owner_name.*] *table_name*)	Name of the tenant for the current row of the specified table.
tenantid_tbl ([owner_name. | *catalog_name.owner_name.*] *table_name*)	Unique ID of the tenant for the current row of the specified table.

IV

ESQL Reference

For details, see the following topics:

- Embedded SQL

17

Embedded SQL

ESQL is provided to help OpenEdge customers transition to the use of other interfaces. For the long term, you should use ODBC or JDBC to access SQL data. These newer interfaces offer better performance and access by many development and reporting tools.

For details, see the following topics:

- ESQL elements and statements
- ESQL elements and statements in Backus Naur Form (BNF)
- Compliance with industry standards

ESQL elements and statements

This topic provides detailed information on OpenEdge SQL statements. A description for each statement provides the following information:

- Definition of the statement
- Syntax of the statement's proper usage
- A code sample that shows how the statement works
- Any associated notes
- Authorization required in order to use the statement
- Related statements

BEGIN-END DECLARE SECTION

Declares variables and types used by the precompiler. Any variables you refer to in an embedded SQL statement must be declared in a DECLARE SECTION. This topic starts with a BEGIN DECLARE SECTION statement and ends with an END DECLARE SECTION statement. Each variable must be declared as a host language data type.

Syntax

```
EXEC SQL BEGIN DECLARE SECTIONhost_lang_typevariable_name ;
  .
  .
  .
EXEC SQL END DECLARE SECTION
```

host_lang_type variable_name ;

A conventional C Language variable declaration. This form of variable declaration conforms to the ANSI standard for the C Language.

Uses the following syntax:

```
{ char | short | long | float | double }
```

Notes

- The C Language type int is not supported by ESQL. Type int maps to 16 or 32 bits, depending on the machine architecture. This can create rounding errors at run time, as values are passed across different machine architectures.
- Variables you declare in a BEGIN-END DECLARE SECTION can be used in C Language statements as if they are declared outside the DECLARE SECTION.
- The scope of variables follows host language scoping rules. The ESQL variables are not visible outside the file in which they are declared.

  DECLARE sections are permissible only where host language declarations are permissible in the host language syntax. This restriction is due to how DECLARE SECTION blocks are translated into the main body of host language declarations.
- Avoid DECLARE sections in header files that are included by more than one source file. This can cause duplicate variables with the same name.
- The form of the variable created by ESQL for each type is specified so that it can be manipulated from host language statements. Declaring variables allows you to use the variables in both host language and embedded SQL statements.

Example

```
EXEC SQL BEGIN DECLARE SECTION ;
      short InvTransNum_v ;
      short Qty_v ;
      short OrderNum_v ;
EXEC SQL END DECLARE SECTION ;
```

Authorization

None

Related statement

Static Array Types

CLOSE

Closing a cursor changes the state of the cursor from open to closed.

Syntax

```
EXEC SQL CLOSE cursor_name ;
```

Parameter

cursor_name

An identifier named earlier in a DECLARE CURSOR statement and an OPEN CURSOR statement.

Notes

- Only a cursor in the open state can be set to the closed state.
- When a transaction ends, any cursors in the open state are automatically set to the closed state.
- When a cursor is in the closed state, you cannot perform FETCH, DELETE, or UPDATE operations using that cursor.
- It is good practice to close cursors explicitly.

Example

```
EXEC SQL CLOSE dyncur ;
EXEC SQL COMMIT WORK ;
```

Authorization

None

Related statements

DELETE, OPEN, FETCH, positioned UPDATE, positioned DELETE

CONNECT

Establishes a connection to a database. Optionally, the CONNECT statement can also specify a name for the connection and a *username* and *password* for authentication.

Syntax

```
CONNECT TO connect_string [ AS connection_name ] [ USER username ] [ USING password ]
 ;
```

Parameters

connect_string

```
{ DEFAULT | db_name | db_type:T:host_name:port_num:db_name }
```

Note: Arguments to CONNECT must be either string literals enclosed in quotation marks or character-string host variables.

connect_string

Specifies to which database to connect. If the CONNECT statement specifies DEFAULT, SQL tries to connect to the environment-defined database, if any. The value of the DB_NAME environment variable specifies the default connect string.

The *connect_string* can be a simple database name or a complete *connect_string*. A complete connect string has the components shown in the following table:

Connect string	Description
db_type	Type of database. The only currently supported database type is *progress*
T	T directs the SQL engine to use the TCP/IP protocol
host_name	Name of the system where the database resides
port_num	Port number to use for the connection
db_name	Name of the database

connection_name

The name of the connection as either a character literal or host variable. If the CONNECT statement omits a connection name, the default is the name of the database. Connection names must be unique.

username

User name for authentication of the connection. SQL verifies the user name against a corresponding password before it connects to the database. The value of the DH_USER environment variable specifies the default user name. If DH_USER is not set, the value of the USER environment variable specifies the default user name.

password

Password for authentication of the connection. SQL verifies the password against a corresponding user name before it connects to the database.

The value of the DH_PASSWD environment variable determines the default password.

Notes

- Arguments to CONNECT must be either string literals enclosed in quotation marks or character string host variables.
- An application can connect to more than one database at a time, with a maximum of 10 connections. However, the application can actually gain access to only one database at a time. The database name specified in the CONNECT statement becomes the active one.
- If an application executes a SQL statement before connecting to a database, an attempt is made to connect to the environment-defined database, if any. If the connection is successful, the SQL statement is executed on that database.

Example

The following example illustrates the CONNECT statement:

```
CONNECT TO "salesdb" AS "sales_conn";
CONNECT TO "progress:T:localhost:custdb" AS "cust_conn";
CONNECT TO DEFAULT;
```

- The first statement shown connects to the salesdb database on the local system.
- The second statement connects to the custdb database on the local system.
- The last statement connects to the environment-defined database by default.

Authorization

None

Related statements

DISCONNECT, SET CONNECTION

DECLARE CURSOR

Associates a cursor with a static query or a prepared dynamic query statement. The query or the prepared statement can have references to host variables.

Syntax

```
DECLARE cursor_name CURSOR FOR
  {query_expression [ ORDER BY clause ]
  [ FOR UPDATE clause ] | prepared_statement_name } ;
```

Parameters

cursor_name

A name you assign to the cursor. The name must meet the requirements for an identifier.

query_expression **[** ORDER BY *clause* **]** **[** FOR UPDATE *clause* **]**

A complete query expression.

prepared_statement_name

The name assigned to a prepared SQL statement in an earlier PREPARE statement.

Notes

- You must declare a cursor before any OPEN, FETCH, or CLOSE statement.
- The scope of the cursor declaration is the entire source file in which it is declared. The operations on the cursor, such as OPEN, CLOSE, and FETCH statements, can occur only within the same compilation unit as the cursor declaration.
- The use of a cursor allows the execution of the positioned forms of the UPDATE and DELETE statements.
- If the DECLARE statement corresponds to a static SQL statement with parameter references:
  - The DECLARE statement must be executed before each execution of an OPEN statement for the same cursor.
  - The DECLARE statement and the OPEN statement that follows must occur within the same transaction within the same task.
  - If the statement contains parameter references to automatic variables or function arguments, the DECLARE statement and the following OPEN statement for the same cursor must occur within the same C function.

Example

```
EXEC SQL WHENEVER SQLERROR GOTO selerr ;
EXEC SQL DECLARE stcur CURSOR FOR
     SELECT InvTransNum, Qty, OrderNum FROM PUB.InventoryTrans ;
EXEC SQL OPEN stcur ;
EXEC SQL WHENEVER NOT FOUND GOTO seldone ;
```

```
EXEC SQL WHENEVER SQLERROR GOTO selerr ;
EXEC SQL PREPARE stmtid from :sel_stmt_v ;
EXEC SQL DECLARE dyncur CURSOR FOR stmtid ;
EXEC SQL OPEN dyncur ;
EXEC SQL WHENEVER NOT FOUND GOTO seldone ;
```

Authorization

None

Related statements

PREPARE, OPEN, FETCH, CLOSE SELECT

DESCRIBE

Writes information about a prepared statement to the SQL Descriptor Area (SQLDA). You use a DESCRIBE statement in a series of steps that allows a program to accept SQL statements at run time. Dynamically generated statements are not part of a program's source code; they are generated at run time.

There are two forms of the DESCRIBE statement:

- The DESCRIBE BIND VARIABLES statement writes information about input variables in an expression to an SQLDA. These variables can be substitution variable names or parameter markers.
- The DESCRIBE SELECT LIST statement writes information about select list items in a prepared SELECT statement to an SQLDA.

Syntax

```
DESCRIBE [ BIND VARIABLES | SELECT LIST ]
  FOR statement_name  INTO input_sqlda_name ;
```

The SQLDA is a host language data structure used in dynamic SQL processing. DESCRIBE statements write information about the number, data types, and sizes of input variables or select list items to SQLDA structures. Program logic then processes that information to allocate storage. OPEN, EXECUTE, and FETCH statements read the SQLDA structures for the addresses of the allocated storage.

DESCRIBE BIND VARIABLES

Writes information about any input variables in the prepared statement to an input SQLDA structure.

Syntax

```
DESCRIBE BIND VARIABLES FOR statement_name INTO input_sqlda_name ;
```

Parameters

statement_name

The name of an input SQL statement to be processed using dynamic SQL steps. Typically, this is the same *statement_name* used in the PREPARE statement.

input_sqlda_name

The name of the SQLDA structure to which DESCRIBE will write information about input variables. Input variables represent values supplied to INSERT and UPDATE statements at run time, and to predicates in DELETE, UPDATE, and SELECT statements at run time.

To utilize the DESCRIBE BIND VARIABLES statement in your application, issue statements in the following order:

1. PREPARE
2. DESCRIBE BIND VARIABLES
3. EXECUTE or OPEN CURSOR

The DESCRIBE BIND VARIABLES statement writes the number of input variables to the sqld_nvars field of the SQLDA. If the sqld_size field of the SQLDA is not equal to or greater than this number, DESCRIBE writes the value as a negative number to sqld_nvars. Design your application to check sqld_nvars for a negative number to determine if a particular SQLDA is large enough to process the current input statement.

Input variables in dynamic SQL statements are identified by parameter markers or as substitution names.

Authorization

None

Related statements

PREPARE, DECLARE CURSOR, OPEN, FETCH, CLOSE

DESCRIBE SELECT LIST

Writes information about select list items in a prepared SELECT statement to an output SQLDA structure.

Syntax

```
DESCRIBE SELECT LIST FOR statement_name INTO output_sqlda_name ;
```

Parameters

statement_name

The name of a SELECT statement to be processed using dynamic SQL steps. Typically, this is the same *statement_name* as in the PREPARE statement.

output_sqlda_name

The name of the SQLDA structure to which DESCRIBE will write information about select list items.

Notes

Select list items are column names and expressions in a SELECT statement. A FETCH statement writes the values returned by a SELECT statement to the addresses stored in an output SQLDA.

To utilize the DESCRIBE SELECT LIST statement in your application, issue statements in the following order:

1. DECLARE CURSOR
2. PREPARE
3. OPEN
4. DESCRIBE SELECT LIST
5. FETCH

A DESCRIBE SELECT LIST statement writes the number of select list items to the sqld_nvars field of an output SQLDA. If the sqld_size field of the SQLDA is not equal to or greater than this number, DESCRIBE writes the value as a negative number to sqld_nvars. Design your application to check sqld_nvars for a negative number to determine if a particular output SQLDA is large enough to process the current SELECT statement.

Authorization

None

Related statements

PREPARE, DECLARE CURSOR, OPEN, FETCH, CLOSE

DISCONNECT

Terminates the connection between an application and the database to which it is connected.

Syntax

```
DISCONNECT { 'connection_name' | CURRENT | ALL | DEFAULT } ;
```

Parameters

connection_name

The name of the connection as either a character literal or host variable.

CURRENT

Disconnects the current connection.

ALL

Disconnects all established connections.

DEFAULT

Disconnects the connection to the default database.

Notes

- When you specify DISCONNECT*connection_name* or DISCONNECT CURRENT and there is also an established connection to the DEFAULT database, the connection to the DEFAULT database becomes the current connection. If there is no DEFAULT database, there is no current connection after the SQL engine processes the DISCONNECT.
- The DISCONNECT DEFAULT statement terminates the connection to the DEFAULT database. If this connection is the current connection, there is no current connection after this DISCONNECT statement is executed.

Example

This example illustrates CONNECT TO AS*connection_name* and DISCONNECT*connection_name*:

```
EXEC SQL
     CONNECT TO 'progress:T:localhost:6745:salesdb' AS 'conn_1' ;
/*
**  C Language and embedded SQL application processing against the
**  database in the connect_string
*/
   .
   .
   .
EXEC SQL
     DISCONNECT 'conn_1' ;
```

The following example illustrates CONNECT TO DEFAULT and DISCONNECT DEFAULT:

```
EXEC SQL
    CONNECT TO DEFAULT ;
/*
**  C Language and embedded SQL application processing against the
**  database in the connect_string
*/
   .
   .
   .
EXEC SQL
    DISCONNECT DEFAULT ;
```

After you issue DISCONNECT ALL there is no current connection. The following example disconnects all database connections:

```
EXEC SQL
     DISCONNECT ALL;
```

The following example illustrates the CONNECT, SET CONNECTION, and DISCONNECT statements in combination using these steps:

1. CONNECT TO *connect_string* AS *connection_name*, which establishes a *connect_string* connection to the database in the *connect_string*; the connection has the name 'conn_1'.
2. CONNECT TO DEFAULT, which establishes a connection to the DEFAULT database and sets this connection current.
3. DISCONNECT DEFAULT, which disconnects the connection to the DEFAULT database.
4. SET CONNECTION *connection_name*, which sets the 'conn_1' connection current
5. DISCONNECT CURRENT, which disconnects the 'conn_1' connection.

```
/*
**  1. CONNECT TO 'connect_string'
*/
EXEC SQL
     CONNECT TO 'progress:T:localhost:6745:salesdb' AS 'conn_1' ;
/*
**  2. CONNECT TO DEFAULT. This suspends the conn_1 connection
**     and sets the DEFAULT connection current
*/
EXEC SQL
     CONNECT TO DEFAULT ;
/*
** Application processing against the DEFAULT database
*/
   .
   .
   .
/*
**  3. DISCONNECT DEFAULT
*/
EXEC SQL
     DISCONNECT DEFAULT ;
/*
**  4. Set the first connection, conn_1, current
*/
EXEC SQL
     SET CONNECTION conn_1 ;
/*
** Application processing against the database in the connect_string
*/
   .
   .
   .
/*
**  5. DISCONNECT the conn_1 connection, which is the current connection.
*/
EXEC SQL
     DISCONNECT CURRENT ;
```

Authorization

None

Related statements

CONNECT, SET CONNECTION

EXEC SQL delimiter

In C Language programs, you must precede embedded SQL statements with the EXEC SQL delimiter so that the precompiler can distinguish statements from the host language statements.

Note: Constructs within a BEGIN-END DECLARE SECTION do not require the EXEC SQL delimiter.

Syntax

```
EXEC SQL sql_statement ;
```

Parameter

sql_statement

An SQL statement to be processed by the ESQL precompiler. You must terminate each SQL statement with a semicolon to mark the end of the statement.

Notes

In general, the ESQL precompiler does not parse host language statements and therefore does not detect any syntax or semantic errors in host language statements. The exceptions to this rule are:

- Recognition of host language blocks. The precompiler recognizes host language blocks in order to determine the scope of variables and types.
- Constants defined with the #define preprocessor command. To evaluate these constants, the ESQL precompiler invokes the C language preprocessor before beginning embedded SQL processing.

Example

```
EXEC SQL WHENEVER SQLERROR GOTO selerr ;
EXEC SQL PREPARE stmtid from :sel_stmt_v ;
EXEC SQL DECLARE dyncur CURSOR FOR stmtid ;
EXEC SQL OPEN dyncur ;
EXEC SQL WHENEVER NOT FOUND GOTO seldone ;
```

Authorization

None

EXECUTE

Executes the statement specified in *statement_name*.

Syntax

```
EXECUTE statement_name [ USING
     {[ SQL ] DESCRIPTOR structure_name | :host_variable
      [[ INDICATOR ]:ind_variable] , ...}] ;
```

Parameters

statement_name

Name of the prepared SQL statement.

structure_name

Name of an SQL descriptor area (SQLDA).

Notes

- A statement must be processed with a PREPARE statement before it can be processed with an EXECUTE statement.
- A prepared statement can be executed multiple times in the same transaction. Typically each call to the EXECUTE statement supplies a different set of host variables.
- If there is no DESCRIPTOR in the USING clause, the EXECUTE statement is restricted to the number of variables specified in the host variable list. The number and type of the variables must be known at compile time. The host variables must be declared in the DECLARE SECTION before they can be used in the USING clause of the EXECUTE statement.
- If there is a DESCRIPTOR in the USING clause, the program can allocate space for the input host variables at run time.

Example

```
/*
** Process the non-SELECT input statement
**    PREPARE the statement
**    EXECUTE the prepared statement
**    COMMIT WORK
*/
      EXEC SQL PREPARE dynstmt FROM :sql_stmt_v ;
      EXEC SQL EXECUTE dynstmt ;
      EXEC SQL COMMIT WORK ;
```

Related statements

EXECUTE IMMEDIATE, PREPARE

EXECUTE IMMEDIATE

Executes the statement specified in a *statement_string* or *host_variable*.

Syntax

```
EXECUTE IMMEDIATE { statement_string | host_variable };
```

Parameters

statement_string

The character string form of the statement.

host_variable

The host variable name which passes values between the database and the application program.

Notes

- The character string form of the statement is referred to as a statement string. An EXECUTE IMMEDIATE statement accepts either a statement string or a host variable as input.
- A statement string must not contain host variable references or parameter markers.
- A statement string must not begin with EXEC SQL delimiter and must not end with a semicolon.
- When an EXECUTE IMMEDIATE statement is executed, the SQL engine parses the statement and checks it for errors. Any error in the execution of the statement is reported in the SQLCA.
- If the same SQL statement is to be executed multiple times, it is more efficient to use PREPARE and EXECUTE statements, rather than an EXECUTE IMMEDIATE statement.

Related statement

EXECUTE

FETCH

Moves the position of the cursor to the next row of the active set and fetches the column values of the current row into the specified host variables.

Syntax

```
FETCH cursor_name{ USING SQL DESCRIPTOR structure_name
 |INTO:host_var_ref[[ INDICATOR ] :ind_var_ref] , ...} ;
```

Parameters

cursor_name

A name identified in an earlier DECLARE CURSOR statement and an OPEN CURSOR statement.

USING SQL DESCRIPTOR *structure_name*

Directs the SQL engine to FETCH data into storage addressed by an SQLDA structure.

INTO :*host_var_ref* [[INDICATOR] :*ind_var_ref*]

Directs the SQL engine to FETCH data into the identified host variables, and to set values in the identified indicator variables.

Notes

- A FETCH operation requires that the cursor be open.
- The positioning of the cursor for each FETCH operation is as follows:
  - The first time you execute a FETCH statement after opening the cursor, the cursor is positioned to the first row of the active set.
  - Subsequent FETCH operations advance the cursor position in the active set. The next row becomes the current row.
  - When the current row is deleted using a positioned DELETE statement, the cursor is positioned before the row after the deleted row in the active set.
- The cursor can only be moved forward in the active set by executing FETCH statements. To move the cursor to the beginning of the active set, you must CLOSE the cursor and OPEN it again.
- If the cursor is positioned on the last row of the active set or if the active set does not contain any rows, executing a FETCH will return the status code SQL_NOT_FOUND in the SQLDA.
- After a successful FETCH, the total row count fetched so far for this cursor is returned in sqlca.sqlerrd[2]. The count is set to zero after an OPEN cursor operation.
- You can FETCH multiple rows in one FETCH operation by using array variables in the INTO clause. The SQL_NOT_FOUND status code is returned in the SQLCA when the end of the active set is reached, even if the current FETCH statement returns one or more rows.
- If you use array variables in a FETCH statement, the array sizes are set to the number of rows fetched after the FETCH statement is executed.

Example

```
/*
**  One way to limit the number of rows returned is to
**  set a new value for "j" here. As supplied in the SPORTS2000 database,
**  the PUB.InventoryTrans table contains 75 rows.
*/
      j = 100;
      for (i = 0; i < j; i++)
           {
           EXEC SQL FETCH dyncur INTO
             :int_p1_v, :int_p2_v, :char_p_v ;
           if (i == 0)
               {
               printf (" 1st col  2nd col  3rd col");
               printf (" -------  -------  --------");
               }
           printf (" %d  %d  %s ",
                   int_p1_v, int_p2_v, char_p_v) ;
           }
```

Authorization

None

Related statements

DECLARE CURSOR, OPEN, CLOSE

GET DIAGNOSTICS

Retrieves information about the execution of the previous SQL statement from the SQL diagnostics area. The diagnostics area is a data structure that contains information about the execution status of the most recent SQL statement. Specifically, GET DIAGNOSTICS extracts information about the SQL statement as a whole from the SQL diagnostics area's header component.

Note: The GET DIAGNOSTICS EXCEPTION number extracts detail information.

Syntax

```
GET DIAGNOSTICS
  :param = header_info_item [ , :param = header_info_item ] , ... ;
```

Parameters

param

A host-language variable to receive the information returned by the GET DIAGNOSTICS statement. The host-language program must declare a *param* compatible with the SQL data type of the information item.

header_info_item

One of the following keywords, which returns associated information about the diagnostics area or the SQL statement:

```
NUMBER | MORE | COMMAND_FUNCTION | DYNAMIC_FUNCTION | ROW_COUNT
```

NUMBER

The number of detail areas in the diagnostics area. Currently, NUMBER is always 1. NUMBER is type NUMERIC with a scale of 0.

MORE

A one-character string with a value of Y (all conditions are detailed in the diagnostics area) or N (all conditions are not detailed) that tells whether the diagnostics area contains information on all the conditions resulting from the statement.

COMMAND_FUNCTION

Contains the character-string code for the statement (as specified in the SQL standard), if the statements is a static SQL statement. If the statement is a dynamic statement, contains the character string EXECUTE or EXECUTE IMMEDIATE.

DYNAMIC_FUNCTION

Contains the character-string code for the statement (as specified in the SQL standard). For dynamic SQL statements only (as indicated by EXECUTE or EXECUTE IMMEDIATE in the COMMAND_FUNCTION item).

ROW_COUNT

The number of rows affected by the SQL statement.

Example

The GET DIAGNOSTICS example extracts header information about the last SQL statement executed. The information is assigned to host variables that are defined in the DECLARE SECTION of an embedded SQL program, as shown in the following example:

```
GET DIAGNOSTICS :num = NUMBER, :cmdfunc = COMMAND_FUNCTION ;
```

The GET DIAGNOSTICS statement itself does not affect the contents of the diagnostics area. This means applications can issue multiple GET DIAGNOSTICS statements to retrieve different items of information about the same SQL statement.

Related statements

GET DIAGNOSTICS EXCEPTION, WHENEVER

GET DIAGNOSTICS EXCEPTION

Retrieves information about the execution of the previous SQL statement from the SQL diagnostics area. The diagnostics area is a data structure that contains information about the execution status of the most recent SQL statement. Specifically, GET DIAGNOSTICS EXCEPTION extracts information about the SQL statement as a whole from the SQL diagnostics area's detail component.

The detail area contains information for a particular condition (an error, warning, or success condition) associated with execution of the last SQL statement. The diagnostics area can potentially contain multiple detail areas corresponding to multiple conditions generated by the SQL statement described by the header. The SQL diagnostics area currently supports only one detail area.

Note: The GET DIAGNOSTICS statement extracts header information.

Syntax

```
GET DIAGNOSTICS EXCEPTION number  :param = detail_info_item [, :param =
detail_info_item] , ... ;
```

Parameters

EXCEPTION *number*

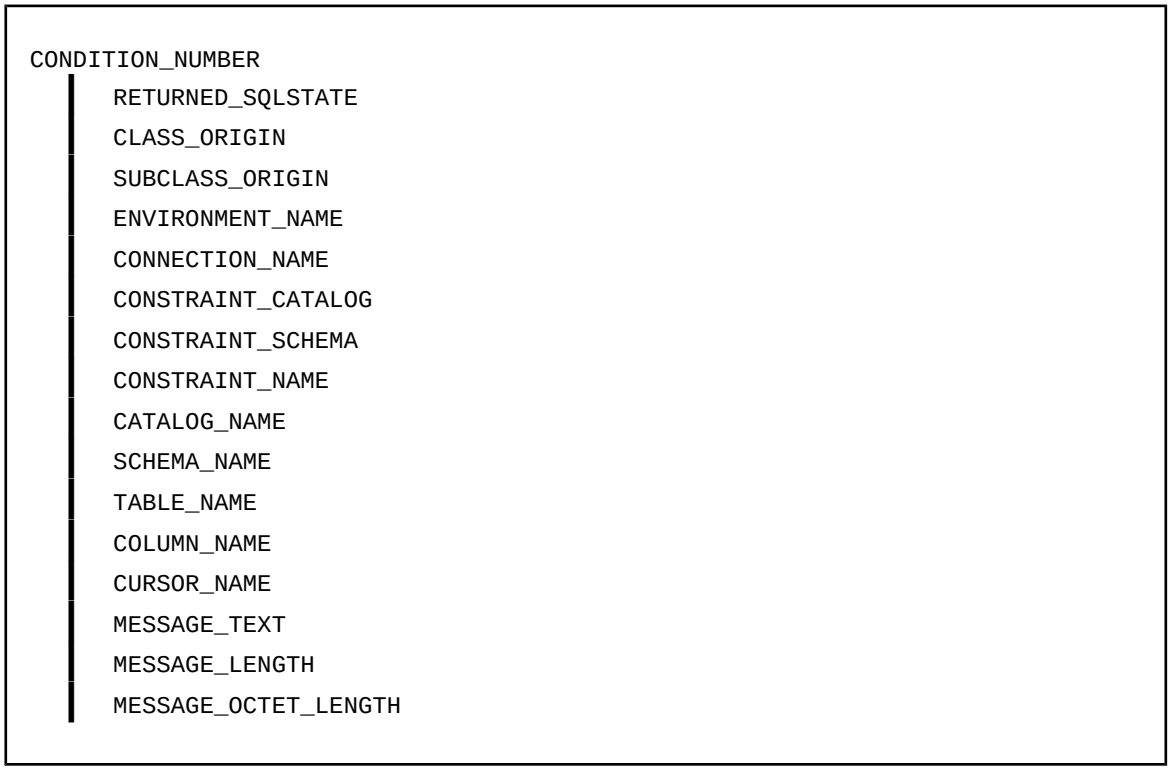
Specifies that GET DIAGNOSTICS EXCEPTION extracts detail information. *number* specifies which of multiple detail areas GET DIAGNOSTICS extracts. Currently, *number* must be the integer 1.

param

Receives the information returned by the GET DIAGNOSTICS EXCEPTION statement. The host-language program must declare a *param* compatible with the SQL data type of the information item.

detail_info_item

One of the following keywords, which returns associated information about the particular error condition:



CONDITION_NUMBER

The sequence of this detail area in the diagnostics area. Currently, CONDITION_NUMBER is always 1.

RETURNED_SQLSTATE

The SQLSTATE value that corresponds to the condition.

CLASS_ORIGIN

The general type of error. For example, connection exception or data exception.

SUBCLASS_ORIGIN

The specific error. Usually the same as the message text.

ENVIRONMENT_NAME

Not currently supported.

CONNECTION_NAME

Not currently supported.

CONSTRAINT_CATALOG

Not currently supported.

CONSTRAINT_SCHEMA

Not currently supported.

CONSTRAINT_NAME

Not currently supported.

CATALOG_NAME

Not currently supported.

SCHEMA_NAME

Not currently supported.

TABLE_NAME

The name of the table, if the error condition involves a table.

COLUMN_NAME

The name of the affected columns, if the error condition involves a column.

CURSOR_NAME

Not currently supported.

MESSAGE_TEXT

The associated message text for the error condition.

MESSAGE_LENGTH

The length in characters of the message in the MESSAGE_LENGTH item.

MESSAGE_OCTET_LENGTH

Not currently supported.

Notes

The GET DIAGNOSTICS statement itself does not affect the contents of the diagnostics area. This means applications can issue multiple GET DIAGNOSTICS statements to retrieve different items of information about the same SQL statement.

Example

The GET DIAGNOSTICS EXCEPTION example extracts detailed information into host variables that are defined in the DECLARE SECTION of an embedded SQL program:

```
GET DIAGNOSTICS EXCEPTION :num :sstate = RETURNED_SQLSTATE,
     :msgtxt = MESSAGE_TEXT ;
```

Related statements

GET DIAGNOSTICS, WHENEVER

OPEN

Executes a prepared SQL query associated with a cursor and creates a result set composed of the rows that satisfy the query. This set of rows is called the active set.

Syntax

```
OPEN cursor_name [ USING { [ SQL ] DESCRIPTOR structure_name
  | :host_variable [ [ INDICATOR ] :ind_variable ] , ... } ] ;
```

Parameters

cursor_name

An identifier named in an earlier DECLARE CURSOR statement.

USING [SQL] DESCRIPTOR *structure_name*

Directs the SQL engine to create the result set in storage addressed by the identified SQLDA structure.

USING :*host_variable* [[INDICATOR] :*ind_variable*]

Directs the SQL engine to create the result set in storage addressed by host variables.

Notes

- Executing an OPEN cursor statement sets the cursor to the open state.
- After the OPEN cursor statement is executed, the cursor is positioned just before the first row of the active set.
- For a single execution of an OPEN cursor statement, the active set does not change and the host variables are not re-examined.
- If you elect to retrieve a new active set and a host variable value has changed, you must CLOSE the cursor and OPEN it again.
- Execution of a COMMIT statement or ROLLBACK statement implicitly closes the cursors that have been opened in the current transaction.

- It is good practice to CLOSE cursors explicitly.
- When a cursor is in the open state, executing an OPEN statement on that cursor results in an error.
- If a DECLARE cursor statement is associated with a static SQL statement containing parameter markers, the following requirements apply:
  - You must execute the DECLARE statement before executing the OPEN statement for that cursor.
  - The DECLARE cursor statement and the OPEN statement for the same cursor must occur in the same transaction.
  - If the statement contains parameter markers for stack variables, the DECLARE cursor statement and the following OPEN statement for the same cursor must occur in the same C Language function.

Example

```
/*
**    5.  Name WHENEVER routine to handle SQLERROR.
**
**    6.  DECLARE cursor for the SELECT statement.
**        NOTE: You must set input parameter values before OPEN CURSOR.
**        The static query in this program does not have input parameters.
**
**    7.  OPEN the cursor.
**        NOTE: For static statements, if a DECLARE CURSOR
**              statement contains references to automatic variables,
**              the OPEN CURSOR statement must be in the same C function.
**
**    8.  Name WHENEVER routine to handle NOT FOUND condition.
*/

      EXEC SQL WHENEVER SQLERROR GOTO selerr ;
      EXEC SQL DECLARE stcur CURSOR FOR
            SELECT InvTransNum, Qty,
            OrderNum FROM PUB.InventoryTrans ;
      EXEC SQL OPEN stcur ;
      EXEC SQL WHENEVER NOT FOUND GOTO seldone ;
```

Authorization

Must have DBA privilege of SELECT privilege on all the tables and views referenced in the SELECT statement associated with the cursor.

Related statements

DECLARE CURSOR, CLOSE, FETCH, positioned UPDATE, positioned DELETE

PREPARE

Parses and assigns a name to an ad hoc or dynamically generated SQL statement for execution. You use a PREPARE statement in a series of steps that allows a program to accept or generate SQL statements at run time.

Syntax

```
PREPARE statement_name FROM statement_string ;
```

Parameters

statement_name

A name for the dynamically generated statement. DESCRIBE, EXECUTE, and DECLARE CURSOR statements refer to this *statement_name*. A *statement_name* must be unique in a program.

statement_string

Specifies the SQL statement to be prepared for dynamic execution. You can use either the name of a C Language string variable containing the SQL statement, or you can specify the SQL statement as a quoted literal. If there is an SQL syntax error, the PREPARE statement returns an error in the SQLCA.

```
{ :host_variable | quoted_literal }
```

Notes

- A statement string can have one or more references to input variables. These variables represent values supplied at run time to:
  - INSERT and UPDATE statements
  - Predicates in DELETE, UPDATE, and SELECT statements
- A program supplies an input variable to a PREPARE statement either as a substitution name or as a parameter marker. For example:
  - A substitution name is a name preceded by a colon (:) in a statement string. This name does not refer to a C Language variable, but acts only as a placeholder for input variables.
  - A parameter marker is a question mark (?) in the statement string, serving as a placeholder for input variables.
- The USING clauses of EXECUTE and OPEN statements identify host language storage. The values in this storage expand a statement string, replacing a substitution name or a parameter marker. You can design your program to execute the same prepared statement many times in a transaction, supplying different values for input variables for each execution. If you COMMIT or ROLLBACK the transaction, you must PREPARE the statement string again.

Example

The first example is a code fragment from the DynUpd function in sample program 3DynUpd.pc, which illustrates dynamic processing of an UPDATE statement:

```
/*
** Process a dynamic non-SELECT input statement
**    PREPARE the statement
**    EXECUTE the prepared statement
**    COMMIT WORK
*/
EXEC SQL PREPARE dynstmt FROM :sql_stmt_v ;
EXEC SQL EXECUTE dynstmt ;
EXEC SQL COMMIT WORK ;
```

This example is a code fragment from the DynSel function in sample program 4DynSel.pc, which illustrates dynamic processing of a SELECT statement:

```
/*
**    PREPARE a the dynamic SELECT statement.
**    DECLARE cursor for the prepared SELECT statement.
**    NOTE: You must set input parameter values before OPEN CURSOR.
**    If your query has input parameters, you must define them in
**    the DECLARE SECTION.
**    OPEN the declared cursor.
**    NOTE: For static statements, if a DECLARE CURSOR
**    statement contains references to automatic variables,
**    the OPEN CURSOR statement must be in the same C function.
**
**    Name WHENEVER routine for NOT FOUND condition.
**    FETCH a row and print results until no more rows.
*/
EXEC SQL PREPARE stmtid from :sel_stmt_v ;
EXEC SQL DECLARE dyncur CURSOR FOR stmtid ;
EXEC SQL OPEN dyncur ;
EXEC SQL WHENEVER NOT FOUND GOTO seldone ;
```

Authorization

Must have DBA privileges or authorization for the SQL statement being used.

Related statements

EXECUTE, OPEN, CLOSE, FETCH

SET CONNECTION

Switches the application from one established connection to another. This resumes the connection associated with the specified *connection_name*, restoring the context of that database connection to the same state it was in when suspended.

Syntax

```
SET CONNECTION { 'connection_name' | DEFAULT } ;
```

Parameters

connection_name

The name of the connection as either a character literal or host variable. If the SET CONNECTION statement omits a connection name, the default is the name of the database. Connection names must be unique.

DEFAULT

Sets the DEFAULT connection as the current connection.

Example

The first example shows how to establish a database as the current database:

```
EXEC SQL
     SET CONNECTION 'conn_1' ;
```

The SET CONNECTION command sets the database associated with the connection named conn_1 to the status of current database. The connection named conn_1 must be associated with an established connection. Use SET CONNECTION DEFAULT to set current the database associated with the DEFAULT connection. In this example, the statement suspends the conn_1 connection, which had been current:

```
EXEC SQL
     SET CONNECTION DEFAULT ;
```

See also the last example for the DISCONNECT statement, which illustrates the CONNECT, SET CONNECTION, and DISCONNECT statements in combination.

Authorization

None

Related statements

CONNECT, DISCONNECT

WHENEVER

Specifies actions for three SQL run-time exceptions.

Syntax

```
WHENEVER
  { NOT FOUND | SQLERROR | SQLWARNING }
  { STOP | CONTINUE |{ GOTO | GO TO }host_lang_label} ;
```

Parameters

{ NOT FOUND | SQLERROR | SQLWARNING }

- The NOT FOUND exception is set when sqlca.sqlcode is set to SQL_NOT_FOUND.
- The SQLERROR exception is set when sqlca.sqlcode is set to a negative value.
- The SQLWARNING exception is set when sqlca.sqlwarn[0] is set to W after a statement is executed.

{ STOP | CONTINUE | GOTO | GO TO }*host_lang_label*}

- The STOP exception results in the ESQL program stopping execution.
- The CONTINUE exception results in the ESQL program continuing execution. The default exception is to CONTINUE.
- GOTO | GO *host_lang_label* results in the ESQL program execution to branch to the statement corresponding to the *host_lang_label*.

Notes

- You can place multiple WHENEVER statements for the same exception in a source file. Each WHENEVER statement overrides the previous WHENEVER statement specified for the same exception.
- Correct operation of a WHENEVER statement with a GOTO*host_language_label* or a GO TO*host_language_label* is subject to the scoping rules of the C Language. The *host_language_label* must be within the scope of all SQL statements for which the action is active. The GO TO or GOTO action is active starting from the corresponding WHENEVER statement until another WHENEVER statement for the same exception, or until end of the file.

Example

```
/*
**  Name WHENEVER routine to handle SQLERROR condition.
*/
EXEC SQL WHENEVER SQLERROR GOTO mainerr ;
```

```
/*
**    Name WHENEVER routines to handle NOT FOUND and SQLERROR
*/
EXEC SQL WHENEVER SQLERROR GOTO nodyn ;
EXEC SQL WHENEVER NOT FOUND GOTO nodyn ;
```

Authorization

None

Related statements

FETCH

ESQL elements and statements in Backus Naur Form (BNF)

BEGIN-END DECLARE SECTION

```
begin declare section ::=
EXEC SQL BEGIN DECLARE SECTIONhost_lang_typevariable_name ;
  .
  .
  .
END DECLARE SECTION ::=
EXEC SQL END DECLARE SECTION
```

Host Language Type

```
host language type ::=
{   char
  | short
  | long
  | float
  | double}
```

CLOSE

```
close ::=
EXEC SQL CLOSE cursor_name ;
```

CONNECT

```
connect statement ::=
CONNECT TO connect_string[ AS connection_name]
[ USER user_name][ USING password] ;
```

CONNECT STRING

```
connect_string ::=
{ DEFAULT | db_name | db_type:T:host_name:port_num:db_name }
```

DECLARE CURSOR

```
declare cursor ::=
EXEC SQL DECLARE cursor_name CURSOR FOR
 {query_expr[ ORDER BY clause]
  [ FOR UPDATE clause]|prepared_statement_name};
```

DESCRIBE BIND VARIABLES

```
describe bind variables ::=
EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name  INTO input_sqlda_name ;
```

DESCRIBE SELECT LIST

```
describe select list ::=
EXEC SQL DESCRIBE SELECT LIST FOR statement_name  INTO output_sqlda_name ;
```

DISCONNECT

```
disconnect statement ::=
  DISCONNECT { 'connection_name' | CURRENT | ALL | DEFAULT } ;
```

EXEC SQL

```
EXEC SQL ::=
EXEC SQL  sql_statement ;
```

EXECUTE

```
EXECUTE ::=
 EXEC SQL EXECUTE statement_name[ USING {[ SQL ] DESCRIPTOR structure_name
  |:host_variable[[ INDICATOR ]:ind_variable] , ...}];
```

EXECUTE IMMEDIATE

```
EXECUTE IMMEDIATE ::=
EXEC SQL EXECUTE IMMEDIATE
  { statement_string | host_variable } ;
```

FETCH

```
fetch ::=
EXEC SQL FETCH cursor_name
   { USING SQL DESCRIPTOR structure_name | INTO :host_var_ref
   [[[ INDICATOR ] :ind_var_ref] , ... } ;
```

GET DIAGNOSTICS

```
get diagnostics statement ::=
GET DIAGNOSTICS
  :param = header_info_item[ , :param = header_info_item] , ...;
```

Header Info Item

```
header_info_item ::=
{ NUMBER
  | MORE
  | COMMAND_FUNCTION
  | DYNAMIC_FUNCTION
  | ROW_COUNT }
```

GET DIAGNOSTICS EXCEPTION

```
get diagnostics exception statement ::=
GET DIAGNOSTICS EXCEPTION number
  :param = detail_info_item[, :param = detail_info_item] , ...;
```

Detail Info Item

```
detail_info_item ::=
{ CONDITION_NUMBER
  | RETURNED_SQLSTATE
  | CLASS_ORIGIN
  | SUBCLASS_ORIGIN
  | ENVIRONMENT_NAME
  | CONNECTION_NAME
  | CONSTRAINT_CATALOG
  | CONSTRAINT_SCHEMA
  | CONSTRAINT_NAME
  | CATALOG_NAME
  | SCHEMA_NAME
  | TABLE_NAME
  | COLUMN_NAME
  | CURSOR_NAME
  | MESSAGE_TEXT
  | MESSAGE_LENGTH
  | MESSAGE_OCTET_LENGTH}
```

OPEN

```
open ::=
EXEC SQL OPEN cursor_name [ USING {[ SQL ] DESCRIPTOR structure_name
    | :host_variable[[ INDICATOR ] :ind_variable] , ...} ];
```

PREPARE

```
prepare ::=
EXEC SQL PREPARE statement_name FROM statement_string ;
```

SET CONNECTION

```
set connection statement ::=
  SET CONNECTION { 'connection_name' | DEFAULT } ;
```

SET TRANSACTION ISOLATION LEVEL

```
set transaction isolation level statement ::=
SET TRANSACTION ISOLATION LEVEL isolation_level_name ;
```

ISOLATION LEVEL NAME

```
isolation_level_name ::=
  READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE
```

WHENEVER

```
whenever ::=
 EXEC SQL WHENEVER
  { NOT FOUND | SQLERROR | SQLWARNING }
  { STOP | CONTINUE |{ GOTO | GO TO }host_lang_label} ;
```

Compliance with industry standards

The following table provides details on SQL DDL and DML compliance with industry standards. A check mark indicates compliance.

	SQL	ODBC SQLgrammar	Progress extension	Notes
BEGIN-END DECLARE SECTION	a	—	—	Compliant if C language types usedEmbedded SQL only
CLOSE	a	—	—	Embedded SQL only
DECLARE CURSOR	a	—	*prepared_stmt_name*	Embedded SQL only
DESCRIBE	a	—	—	Embedded SQL only
EXEC SQL	a	—	—	Embedded SQL only

	SQL	ODBC SQLgrammar	Progress extension	Notes
EXECUTE	a	—	—	Embedded SQL only
EXECUTE IMMEDIATE	a	—	—	Embedded SQL only
FETCH	a	—	USING DESCRIPTOR	Embedded SQL only
GET DIAGNOSTICS	—	—	a	Embedded SQL only
OPEN	a	—	USING DESCRIPTOR	Embedded SQL only
PREPARE	a	—	—	Embedded SQL only
ROLLBACK	a	—	—	—
SELECT	a	Extended	FOR UPDATE	—
WHENEVER	a	—	SQLWARNING STOP ACTION	Embedded SQL only

Index

A

ABS function *195*
ACOS function *196*
Add additional index entries here (if needed)
    Nest index terms for sub entries *56*, *67*
ADD_MONTHS function *196*
Aggregate functions *195*
ALTER DATABASE SET
PRO_ENABLE_64BIT_SEQUENCES 'Y' statement *30*
ALTER DATABASE SET PRO_ENABLE_LARGE_KEYS
'Y' statement *29*
ALTER SEQUENCE statement *31*
ALTER TABLE statement *34*
ALTER TENANT statement *69*
ALTER USER statement
    domain_name *71*
Approximate numeric data types *331*
ARRAY data types
    PRO_ELEMENT function *242*
    support *349*
ASCII function *197*
ASIN function *198*
ATAN function *198*
ATAN2 function *199*
AUDIT INSERT statement *72*
AUDIT SET statement *73*
AVG function *200*

B

Backus Naur Form
    data types *351*
    DDL syntax *357*
    DML statements *358*
    ESQL elements and statements *484*
    expressions *351*, *353*
    literals *354*
    query expressions *355*
    search conditions *357*
    SQL elements and statements *351*
Basic predicates
    query expressions in *343*
BEGIN-END DECLARE SECTION *460*
BETWEEN predicate *344*
Bit string data types *333*

C

CALL statement *75*, *79*
CASE function *200*
CAST function *203*
CHAR function *206*
Character data types *329*
Character-string literals *338*
CHR function *207*
Clauses
    COLUMN_LIST *148*
    FOR UPDATE *161*
    FROM *152*
    GROUP BY *154*
    HAVING *156*
    ORDER BY *157*
    WHERE *154*
    WITH *160*
CLOSE statement *461*
COALESCE function *204–205*, *207*
Column constraints *76*
COLUMN_LIST clause *148*
Commit statement *78*
CONCAT function *208*
Concatenation operator *330*
CONNECT statement *462*
Conventional identifiers *323*
CONVERT function (ODBC Compatible) *209*
CONVERT function (Progress Extension) *209*
COS function *210*
COUNT function *211*
Create DOMAIN statement
    SUPER_TENANT super-tenant_name *80*
CREATE DOMAIN Statement *80*
CREATE INDEX statement *79–80*, *83*
CREATE PROCEDURE statement *88*
CREATE SEQUENCE statement *90*
CREATE SUPER-TENANT statement *93*
CREATE SYNONYM statement *93*
CREATE TABLE statement *94*
CREATE TENANT statement *107*
CREATE TRIGGER statement *110*
CREATE USER statement *113*
CREATE VIEW statement *115*
CURDATE function *211*
CURRVAL function *212*
CURTIME function *212*

D

Data types
    ABL *349*
    approximate numeric *331*
    character *329*
    date-time *332*

Data types *(continued)*
definition *328*
exact numeric *330*
OpenEdge SQL *349*
DATABASE function *213*
DatabaseMetaData *412*
Date arithmetic expressions *347*
Date formats *326*
Date-time data types *332*
Date-time formats *325*
Date-time literals *339*
DAYNAME function *214*
DAYOFMONTH function *215*
DAYOFWEEK function *215*
DAYOFYEAR function *216*
DB_NAME function *216*
DECLARE CURSOR statement *464*
DECODE function *217*
DEGREES function *218*
Delimited identifiers *324*
DESCRIBE BIND VARIABLES statement *465*
DESCRIBE SELECT LIST statement *466*
DESCRIBE statement *465*
DhSQLException *384*
DhSQLException.getDiagnostics *384*, *386*
DhSQLResultSet *386*
DhSQLResultSet.insert *386*
DhSQLResultSet.makeNULL *387*
DhSQLResultSet.set *388*
DISCONNECT CATALOG statement *118*
DISCONNECT statement *467*
DROP DOMAIN statement *119*
DROP INDEX statement *119*
DROP PROCEDURE statement *122*
DROP SYNONYM statement *124*
DROP SYNONYM Statement *123*
DROP TABLE statement *124*
DROP TENANT statement *126*
DROP TRIGGER statement *126–127*
DROP USER statement *128*
DROP VIEW statement *129*

E

Embedded SQL *459*
Error Codes
messages *276*
SQLSTATE values *276*
Escape clause
in LIKE predicate *344*
ESQL
compliance with industry standards *488*
statements *459*
Exact numeric data types *330*
EXEC SQL delimiter *470*
EXECUTE IMMEDIATE statement *471*
EXECUTE statement *470*
EXISTS predicate *345*
EXPfunction *218*
Expressions
date arithmetic *347*
numeric arithmetic *346*

F

FETCH statement *472*
FLOOR function *219*
FOR UPDATE clause *161*
Formats
date *326*
date-time *325*
number *324*
time *327*
FROM clause *152*

G

GET DIAGNOSTICS EXCEPTION statement *475*
GET DIAGNOSTICS statement *474*
GRANT statement
WITH GRANT OPTION *130*
GREATEST function *219*
GROUP BY clause *154*

H

HAVING clause *156*
HOUR function *220*

I

Identifiers
conventional *323*
delimited *323*
IFNULL function *221*
IN predicate
query expressions in *346*
Industry standards *367*
INITCAP function *221*
INSERT function *222*
INSERT statement *135*
INSTR function *223*
Internationalization
specific elements
LIKE predicate *344*

J

Java Class Reference
close *382*
err *384*

Java Class Reference *(continued)*
execute *382*
fetch *382*
found *383*
getDiagnostics *383*
getValue *383*
insert *383*
makeNULL *382*
open *382*
rowCount *382*
set *383*
setParam *381*
wasNULL *383*
JDBC conformance
datatype conversion *409*
SQL-92 *409*
supported data types *409*

L

Language elements *323*
LAST_DAY function *226*
LCASE function *226*
LEAST function *227*
LEFT function *227*
LENGTH function *228*
LIKE predicate *344*
Literals
character-string *338*
date-time *339*
numeric *338*
time *340*
timestamp *341*
Literals syntax in BNF *354*
LOCATE function *229*
LOCK TABLE statement *139*
LOG10 function *229*
LOWER function *230*
LPAD function *231*
LTRIM function *231*

M

MAX function *232*
MIN function *233*
MINUTE function *233*
MOD function *234*
MONTH function *234*
MONTHNAME function *235*
MONTHS_BETWEEN function *235*

N

NEXT_DAY function *236*
NOW function *238*
NULL predicate *344*
NULLIF function *238*
Number formats *324*
Numeric arithmetic expressions *346*
Numeric literals *338*, *346*
NVL function *239*

O

ODBC
data types and OpenEdge *425*
date and time functions *453*
numeric functions *450*
Scalar functions *447*
SQLGetInfo *427*
string functions *448*
system functions *455*
OPEN statement *478*
OpenEdge SQL reserved words *269*
OpenEdge SQL system limits *293*
Operators
relational
in quantified predicates *343*
ORDER BY clause *157*
OUTER JOIN predicate *346*

P

PI function *239*
POWER function *240*
Predicates
basic *343*
BETWEEN *344*
EXISTS *345*
IN *346*
LIKE *344*
NULL *344*
OUTER JOIN *346*
quantified *343*
PREFIX function *240*
PREPARE statement *479*
PRO_ELEMENT function *242*

Q

Quantified predicates
query expressions in *343*
QUARTER function *245*
Query Expressions
in basic predicates *343*
in IN predicate *346*
in quantified predicates *343*
Query Expressions syntax in BNF *355*

R

RADIANS function *246*
RAND function *246*
Relational operators
in quantified predicates *343*
REPEAT function *247*
REPLACE function *248*
Required: add at least one index entry *56*, *67*
Reserved words *269*
REVOKE statement
FROM PUBLIC *141*
FROM user_identifier *141*
GRANT OPTION FOR *141*
RESOURCE *141*
RESTRICT | CASCADE *141*
RIGHT function *248*
ROLLBACK statement *145*
ROUND function *249*
ROWID function *250*
RPAD function *251*
RTRIM function *252*

S

Search conditions syntax in BNF *357*
SECOND function *253*
SELECT statement *146*
SET CATALOG statement *162*
SET CONNECTION statement *481*
SET PRO_CONNECT LOG statement *164–166*
SET PRO_SERVER LOG statement *167*
SET SCHEMA statement *171*
SET TRANSACTION ISOLATION LEVEL Statement *172*
SHOW CATALOGS statement *173*
SHOW TENANT statement *178*
SIGN function *253*
SIN function *254*
SQLCursor *389*
SQLCursor.close *390*
SQLCursor.fetch *390*
SQLCursor.found *391*
SQLCursor.getParam *392*
SQLCursor.getValue *393*
SQLCursor.makeNULL *395*
SQLCursor.open *395*
SQLCursor.registerOutParam *396*
SQLCursor.rowCount *397*
SQLCursor.setParam *397*
SQLCursor.wasNULL *398*
SQLGetInfo *427*
SQLIStatement *399*
SQLIStatement.execute *400*
SQLIStatement.makeNULL *401*
SQLIStatement.rowCount *401*
SQLIStatement.setParam *402*
SQLPStatement *403*
SQLPStatement.execute *404*
SQLPStatement.makeNULL *404*
SQLPStatement.rowCount *405*
SQLPStatement.setParam *406*
SQLSTATE values *276*
SQRT function *254*
SUBSTR function *255*
SUBSTRING function (ODBC Compatible) *256*
SUFFIX function *257*
SUM function *258*
SYS_CHK_CONSTRS system table *320*
SYS_CHKCOL_USAGE system table *320*
SYS_KEYCOL_USAGE system table *321*
SYS_REF_CONSTRS system table *321*
SYS_TBL_CONSTRS system table *322*
SYSCALCTABLE system table *301*
SYSCOLAUTH system table *301*
SYSCOLSTAT system table *302*
SYSCOLUMNS core system table *299*
SYSCOLUMNS_FULL system table *303*
SYSDATATYPES system table *304*
SYSDATE function *259*
SYSDATESTAT system table *305*
SYSDBAUTH system table *305*
SYSFLOATSTAT system table *306*
SYSIDXSTAT system table *306*
SYSINDEXES core systemtable *300*
SYSINTSTAT system table *306*
SYSNCHARSTAT system table *301*
SYSNUMSTAT system table *307*
SYSNVARCHARSTAT system table *319*
SYSPROCAUTH system table *309*
SYSPROCBIN system table *307*
SYSPROCCOLUMNS system table *308*
SYSPROCEDURES system table *308*
SYSPROCTEXT system table *309*
SYSREALSTAT system table *310*
SYSSEQAUTH system table *310*
SYSSEQUENCES system table *311*
SYSSYNONYMS system table *312*
SYSTABAUTH system table *312*
SYSTABLES core system table *299*
SYSTABLES_FULL system table *313*
SYSTBLSTAT system table *314*
System Catalog Tables
descriptions *296*
overview *296*
System limits *293*
System tables and descriptions *296*
SYSTIME function *259*
SYSTIMESTAMP function *260*
SYSTIMESTAT system table *315*
SYSTINYINTSTAT system table *315*
SYSTRIGCOLS system table *316*
SYSTRIGGER system table *316*

SYSTSSTAT system table *317*
SYSTSTZSTAT *319*
SYSTSTZSTAT system table *319*
SYSVIEWS system table *320*

T

TAN function *260*
Time formats *327*
Time literals *340*
Timestamp literals *341*
TO_CHAR function *261*
TO_DATE function *262*
TO_NUMBER function *262*
TO_TIME function *263*
TO_TIMESTAMP function *263*
TOP clause *151*
TRANSLATE function *264*

U

UCASE function *265*
UPDATE statement *183*
UPDATE STATISTICS statement *186*
UPPER function *266*
USER function *266*

W

WEEK function *267*
WHENEVER statement *482*
WHERE clause *154*
WITH clause *160*

Y

YEAR function *267*